Asynchronous process calculi: the first-order and higher-order paradigms (Tutorial)

Davide Sangiorgi
INRIA Sophia-Antipolis, France

April 7, 1999

Abstract

We compare the *first-order* and the *higher-order* paradigms for the representation of mobility in process calculi. The prototypical calculus in the first-order paradigm is the π*-calculus*. Here, we focus on an asynchronous π-calculus (Lπ) that may be regarded as the basis of some experimental programming languages (or proposal of programming languages) like Pict, Join, Blue. We extend Lπ so to allow the communication of higher-order values, that is values that may contain processes, and show that the extension does not add expressiveness: the resulting higher-order calculus can be compiled down into Lπ.

This paper is mostly a tutorial. It also contains original contributions. The main one is the full abstraction proof, which, with respect to previous proofs, is simpler and does not rely on certain non-finitary features of the languages such as infinite summation. Another contribution is the study of optimisations of the compilation, with which we are able to handle recursive types and to prove full abstraction also for *strong* behavioural equivalences.

1 Introduction

Mobility is a key concept in distributed computing and concurrent object-oriented languages, whose interest is nowadays rapidly growing as a consequence of the amazing success of worldwide networking. Mobility can also be found in operating systems: examples are a resource that has a single owner at any time but whose ownership can be changed as time passes, or process migration, in which tasks or processes can be exchanged among processors to optimise their load balance.

Although the incarnations of mobility can be quite different, at a first approximation we can group them into two categories: movement of computational entities (objects, processes, that may be exchanged or change site), and movements of communication links. Correspondingly, there are two main approaches to represent mobility in process algebra. In the *higher-order* (or *process-passing*) paradigm, terms of the language (like processes) may be transmitted; γ-calculus [6], Plain CHOCS [39], CML [31], FACILE [13, 41] belong to this category. In the *first-order* (or *name-passing*) paradigm, mobility is achieved by allowing transmission of names (but not arbitrary terms). The π-calculus is the prototypical first-order calculus. Name-passing is simpler than process-passing, both mathematically and pragmatically. The choice of name-passing for π-calculus was motivated—among other things—by the belief that communication of names is enough to model communication involving processes. The formalisation and validation of this claim is a main topic of the present paper.

We carry out our study on *asynchronous-message-passing* calculi. These are calculi where message emission is non-blocking. Asynchronous communications are interesting from the point of view of concurrent and distributed programming languages, because they are easier to implement and they are closer to the communication primitives offered by available distributed systems (see also the *actor* model [15, 2]). In the π-calculus asynchrony is achieved, syntactically, by disallowing output prefix (that is, continuations underneath output messages) and choice. The *asynchronous π-calculus* was first proposed by Honda and Tokoro [17] and Boudol [7]. We add to the asynchronous π-calculus the constraint that the recipient of a channel may only use it in output actions; that is, only the *output capability* of names may be transmitted. This calculus, that we call *Local π* (Lπ), is studied in [23]; very similar calculi are discussed, or at least mentioned, in [18, 5, 19]. Lπ borrows ideas from some experimental programming languages (or proposal of programming languages), most notably Pict [30], Join [12], and Blue [8], and may be regarded as a basis for them (the restriction on output capabilities is not explicit in Pict, but, as we understand from the Pict users, most of Pict programs obey it). A detailed discussion of the differences between Lπ and the ordinary π-calculus (which motivate also the name "Local π") may be found in Section 2.5.

We then extend Lπ with higher-order features. In the resulting calculus, the *Local Higher-Order π-calculus* (LHOπ), parametrised processes, that we call *abstractions*, may be transmitted. An abstraction of type $T \to \mathcal{B}$ needs an argument of type T before becoming a process. The argument itself can be an abstraction; therefore the order of an abstraction, that is the level of arrow nesting in its type, can be arbitrarily high. The order can also be ω, since we allow recursive types.

We show that LHOπ is *representable* in Lπ. This result, besides being an expressiveness result for the first-order paradigm, allows us to use the theory of the first-order calculus to derive proof techniques for the higher-order calculus. Defining proof techniques directly on higher-order calculi is usually hard. For instance, it may be hard to prove that a (labeled) bisimilarity is a congruence relation. In sequential higher-order calculi such congruence results are often proved using Howe's method [20]; but in concurrency this method seems to work only in a limited number of cases (technically, the bisimulation should be in a "delayed late" style, and even in this case local names and scope extrusions may give problems; see [10] for discussions).

But what does it mean that a given source language is representable within a given target language? Typically there are three phases:

1. Formal definition of the semantics of the two languages;

2. Definition of the encoding from the source to the target language;

3. Proof of the correctness of the encoding with respect to the semantics given.

The predominant approach to the semantics of concurrent systems is *operational*. The possible evolutions of processes are inductively described in terms of *transition systems* which then are quotiented by *behavioural equivalence* relations to abstract away from unwanted details. With respect to denotational semantics, the operational method necessitates a different approach to translation-correctness, where behaviours rather than meanings are compared. The choice of the behavioural equivalence should be uniform on the calculi. The behavioural equivalence itself should be interesting, in the sense that it should be a congruence, and the equalities and inequalities that it gives should be justifiable with respect to an abstract notion of observation. Moreover, the encoding should be *fully abstract*, i.e. two source language terms should be equivalent if and only if their translations are equivalent, and *compositional*. To reveal how the encoding modifies transitions, the full abstraction result should be completed by the *operational correspondence* between a term and its translation (i.e., the connection between their transitions).

With the full abstraction demand, we have taken a strong point of view on representability. Full abstraction has two parts: soundness, which says that the equivalence between the translations of two source terms implies that of the source terms themselves; and completeness, which says the converse. While soundness is a necessary property, one might well consider milder forms of completeness. Assuming that the behavioural equivalence is a congruence, completeness means that the translations of equivalent terms are undistinguishable in all contexts of the target language. A milder form of completeness would require, for instance, undistinguishability only in contexts that are translations of source contexts. We ask for full abstraction because we wish to use the target terms in *any* contexts; and when two source terms are indistinguishable, their encodings should *always* be interchangeable. In other words, we want to be able to switch freely between the two calculi. In our case, where the source language is LHOπ and the target language is Lπ, this allows us on the one hand to make use of the abstraction power of LHOπ, which comes from its higher-order nature. On the other hand, to rely on the more elementary and better developed theory of first-order calculi when reasoning over processes; in virtue of the full abstraction result this theory can be lifted up to LHOπ.

This paper is mostly a tutorial. Familiarity with the π-calculus is not required from a reader, but some acquaintance of process calculi will be useful. The paper is based on results and ideas from several sources, especially [39, 40, 32, 34, 23, 30, 24]. However, the paper also contains technical contributions. The main one is the representability proof of process-passing into name-passing. With respect to previous proofs [32], the present one is simpler and does not rely on certain non-finitary features of the languages such as infinite summation. The simplifications are possible because of the syntactic constraints of Lπ. Another contribution is the study of an optimisation of the compilation, with which we are able to handle recursive types and to prove full abstraction also for *strong* behavioural equivalences (in concurrency theory an equivalence is strong if it gives internal actions the same weight as the other actions).

The calculi Lπ and LHOπ that we study are both typed. The type systems we consider are simple: they have recursive types, arrow types, and, as the only base type, unit type. By contrast with usual type systems for the π-calculus, and following Blue and Pict [8, 30], in our type system for LHOπ channels may have a functional type and output is viewed as a special form of application. As a consequence, in LHOπ the output of a value along a channel and the application of a function to a value are realised by the same construct. The results in the paper can be extended to calculi with more sophisticated type systems (for instance, calculi with polyadic communications or with subtyping).

In Sections 2 and 3 we first review operational semantics, behavioural equivalences, basic algebraic theory of Lπ; we then present a type system for the calculus. In Section 4 we extend Lπ so to allow values built out of processes and obtain the calculus LHOπ. In Section 5 we study a compilation of LHOπ into Lπ and prove its full abstraction for weak barbed congruence. In Section 6 we exploit the previous results to derive a labeled bisimulation for LHOπ and prove its congruence. In Section 5 and 6 we omit however recursive types. In Section 7 we introduce an optimisation of the compilation that can handle recursive types and show that the optimised compilation is fully abstract both for weak and strong barbed congruence.

2 The calculus Lπ

The grammars for typed Lπ are in Table 1. By comparison with the calculus in [23], the present one is typed and distinguishes between channels and variables. Channels and variables are *names*.

2.1 Processes

$\mathbf{0}$ denotes the inactive process. An input-prefixed process $a(x).\, P$ waits for a value v at a and then continues as $P\{v/x\}$. An output process $a\langle v\rangle$ makes value v available at a. A composition $P_1 \mid P_2$ is to run two processes in parallel. A restriction $(\boldsymbol{\nu} a : T)\, P$ makes channel a with type T local, or private, to P. A replicated process $!P$ stands for a countable infinite number of copies of P in parallel. The silent prefix $\tau.\, P$, which describes a process that becomes P by doing some internal work, is derivable: $\tau.\, P \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} a : \mathtt{unit} \to \mathcal{B})\, (a\langle\star\rangle \mid a(x).\, P)$ for a, x not free in P.

The output $a\langle v\rangle$ is similar to an application (the application of a to v); the similarity will become clearer in Section 4. In the higher-order extension of Lπ in Section 4, where application is needed, application and output are rendered by the same construct. Having to unify output and application, we have preferred the notation $v\langle w\rangle$, that is more common for applications, over the notations $\overline{v}w$ and $\overline{v}\langle w\rangle$, that are more common for output. We however freely call $v\langle w\rangle$ an output or an application. In $v\langle w\rangle$, the first component v is a value, rather than a name, so that the operational semantics can be given on untyped processes—see discussion at the end of Section 2.3.

With respect to the ordinary π-calculus, Lπ has various syntactic restrictions: no output prefix, no choice, only the output capability of channels can be communicated, and (as a consequence of the output capability constraint)

Grammars:

$a, b, \ldots$			*Channels*
$x, y, \ldots$			*Variables*
			Names
p, q	$::=$	x	variables
	$\mid$	a	channels
			Values
v, w	$::=$	$\star$	unit value
	$\mid$	p	names
			Processes
P, Q, R	$::=$	$\mathbf{0}$	nil
	$\mid$	$P \mid P$	parallel composition
	$\mid$	$(\boldsymbol{\nu} a : T)\, P$	restriction
	$\mid$	$a(x).\, P$	input
	$\mid$	$v\langle w \rangle$	application (or output)
	$\mid$	$!P$	replication
			Types
TYPE	$::=$	$\mathcal{B}$	behaviour (or process) type
	$\mid$	T	value type
			Value types
S, T	$::=$	$\mathtt{unit}$	unit type
	$\mid$	$T \to \mathcal{B}$	arrow type
	$\mid$	X	type variable
	$\mid$	$\mu X.\, T$	recursive type

Table 1: The syntax of typed Lπ

no matching. We discuss and motivate these restrictions in Section 2.5. They have both theoretical and pragmatical consequences.

2.2 Terminologies and notations

We use σ to range over substitutions; for any expression E, we write $E\sigma$ for the result of applying σ to E, with the usual renaming convention to avoid captures. We assign parallel composition the lowest precedence among the operators. Substitutions have precedence over the operators of the language.

An input prefix $a(x).\,P$ and a restriction $(\boldsymbol{\nu} b : T)\,P$ are binders for variable x and channel b, respectively, and give rise in the expected way to the definitions of free and bound channels, variables and names, and of α-conversion. The binding name x of the input prefix has no type annotations because its type can be inferred from that of channel a. Symbol '$=$' denotes equality up to α-conversion. We write $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$ for the free and bound names of P, respectively. In assertions, we say that a name p is *fresh* for expressions $E_1, \ldots, E_n$ (where E_i can be, for instance, a process, an action, or a name) to mean that p does not occur free in any E_i $(1 \leq i \leq n)$. A process P is *closed* if P does not contain free variables. In an input $a(x).\,P$ and an output $v\langle w \rangle$ channel a and value v are in *input subject* and *output subject* position, respectively; w is in *output object* position. We abbreviate $(\boldsymbol{\nu} a_1 : T_1) \ldots (\boldsymbol{\nu} a_n : T_n)\,P$ as $(\boldsymbol{\nu} a_1 : T_1, \ldots, a_n : T_n\,)P$. We use a tilde to indicate a tuple; all notations are extended to tuples componentwise.

2.3 Operational semantics

We give the operational semantics of processes by means of a labeled transition system, which expresses the internal steps that a process can make and the communications with other processes in which it can engage. The set of rules is given in Table 2 along with the symmetric variants of Com and Par. (This is the standard transition system of the π-calculus; for people who have familiarity with the π-calculus it is the *early* transition system.) Transitions are of the form $P \xrightarrow{\mu} P'$, where the label μ ranges over *actions* of the following forms:

τ	interaction (or reduction)
$a[v]$	input of v at a
$(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\,a\langle v \rangle$	output of v at a, extruding bound names $\widetilde{b}$ of type $\widetilde{T}$.

In an output action $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\,a\langle v \rangle$, component $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})$ is used to record those names in v that have been created fresh in P and are not yet known to the environment. In the case of $\mathrm{L}\pi$, $\widetilde{b}$ can consist of at most one name (therefore in rule Open $\widetilde{b}$ is always empty); the notation $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})$ avoids us having two communication rules; moreover $\widetilde{b}$ can consist of more than one name in extensions of $\mathrm{L}\pi$, for instance with product types, or with higher-order values as in Section 4. If $\widetilde{b}$ is empty then $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\,a\langle v \rangle$ is $a\langle v \rangle$ and, similarly, $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T}\,)P$ is P.

In the component $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})$ of an output action we regard $\widetilde{b} : \widetilde{T}$ as a set of a type assignments, and therefore regard two actions $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})a\langle v \rangle$ and $(\boldsymbol{\nu}\widetilde{c} : \widetilde{S})a\langle v \rangle$ equal if $\widetilde{b} : \widetilde{T}$ and $\widetilde{c} : \widetilde{S}$ only differ in the order of their components (of course this is important only in extensions of $\mathrm{L}\pi$ where the cardinality of $\widetilde{b}$ can be greater than one). As a consequence, in the label of the conclusion of rule Open, the position of $c : S$ with respect to $\widetilde{b} : \widetilde{T}$ is irrelevant.

Input and output actions describe possible interactions between P and its environment, while a τ action represents an internal action in which one subprocess of P communicates with another. When a name b is communicated outside of the scope of the $\boldsymbol{\nu}$ that binds it, the $\boldsymbol{\nu}$ must be moved outwards to include both the sender and the receiver. Formally, this is accomplished by moving the original $\boldsymbol{\nu}$ into the label of the output action (rule Open) and then replacing the $\boldsymbol{\nu}$ at the point where the output action meets a corresponding input action and turns into a τ (rule Com). This is known as *scope extrusion.* In the case of input $a[v]$ or output $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\, a\langle v\rangle$, name a is the *subject* of the action. *Weak transitions* are defined as usual: relation $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$; and $\overset{\mu}{\Longrightarrow}$ stands for $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$. Some examples of transitions:

$$\begin{array}{lcl} ((\boldsymbol{\nu} b : T)\, a\langle b\rangle) \mid a(x).\, P & \xrightarrow{\tau} & (\boldsymbol{\nu} b : T)\, (\mathbf{0} \mid P\{^{b}\!/\!_{x}\}) \\ & & \text{if } b \text{ not free in } a(x).\, P \\ (\boldsymbol{\nu} b : T)\, (a\langle b\rangle \mid b(x).\, P) & \xrightarrow{(\boldsymbol{\nu} b:T)\, a\langle b\rangle} & \mathbf{0} \mid b(x).\, P \\ ((\boldsymbol{\nu} b : T)\, a\langle b\rangle) \mid b(x).\, P & \xrightarrow{(\boldsymbol{\nu} c:T)\, a\langle c\rangle} & \mathbf{0} \mid b(x).\, P\ . \end{array}$$

The side condition in rules Com and Par prevents us from inferring transitions such as

$$\begin{array}{rcl} ((\boldsymbol{\nu} b : T)\, a\langle b\rangle) \mid a(x).\, b\langle x\rangle & \xrightarrow{\tau} & (\boldsymbol{\nu} b : T)\, (\mathbf{0} \mid b\langle b\rangle) \\ ((\boldsymbol{\nu} b : T)\, a\langle b\rangle) \mid b(x).\, P & \xrightarrow{(\boldsymbol{\nu} b:T)\, a\langle b\rangle} & \mathbf{0} \mid b(x).\, P\ . \end{array}$$

In the first transition the free occurrence of b is captured by the extrusion of the restriction, violating the rules of static binding. For similar reasons the second transition is dangerous: when composed with an input transition at a (rule Com), the initial free occurrences of b become bound. The operational rules are defined on arbitrary processes—they need not be well-typed or closed. We shall see that only (closed) well-typed processes, however, are guaranteed not to produce a run-time error.

The syntax of Lπ for outputs is $v\langle w\rangle$, rather than $p\langle w\rangle$, because in the latter case the operational rules would not be well-defined: the derivative of a process might not be a process, as in

$$a\langle\star\rangle \mid a(x).\, x\langle c\rangle \xrightarrow{\tau} \mathbf{0} \mid \star\langle c\rangle\ .$$

The problem is on substitutions: $P\{^{v}\!/\!_{x}\}$ might not be a process. This problem does not exist with the present syntax, as shown by the following lemma:

Lemma 2.1 *$P\{^{v}\!/\!_{x}\}$ is a process, for all process P, name x and value v.*

Consequently, the expression E of a judgement $P \xrightarrow{\mu} E$ inferred from the rules of Table 2 is always a process.

2.4 Types

We consider a very basic type system for Lπ, which has unit type as the only basic value type, and has arrow types and recursive types (recursive types are

$$\text{ALPHA}\ \frac{P\ \alpha\text{-convertible to } P' \qquad P' \xrightarrow{\mu} P''}{P \xrightarrow{\mu} P''}$$

$$\text{INP}\ \frac{}{a(x).\, P \xrightarrow{a[v]} P\{v/x\}} \qquad \text{PAR}\ \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2} \text{bn}(\mu) \cap \text{fn}(P_2) = \emptyset$$

$$\text{OUT}\ \frac{}{a\langle v\rangle \xrightarrow{a\langle v\rangle} \mathbf{0}} \qquad \text{COM}\ \frac{P_1 \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\, a\langle v\rangle} P_1' \quad P_2 \xrightarrow{a[v]} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\,(P_1' \mid P_2')}\ \widetilde{b} \cap \text{fn}(P_2) = \emptyset$$

$$\text{REP}\ \frac{P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} !P \mid P'} \qquad \text{RES}\ \frac{P \xrightarrow{\mu} P'}{(\boldsymbol{\nu}c : T)\, P \xrightarrow{\mu} (\boldsymbol{\nu}c : T)\, P'}\ c \notin (\text{fn}(\mu) \cup \text{bn}(\mu))$$

$$\text{OPEN}\ \frac{P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\, a\langle v\rangle} P'}{(\boldsymbol{\nu}c : S)\, P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T}, c:S\,)\, a\langle v\rangle} P'}\ c \in \text{fn}(v) - \widetilde{b}$$

Table 2: Transition rules for the π-calculus.

---

necessary for writing output processes like $a\langle a\rangle$, see Example 2.5). $\mathcal{B}$ is the type of behaviours (or process type). In a recursive type $\mu X.\, S$, variable X must be guarded in S, i.e., the free occurrences of X in S must occur underneath an arrow type; this rules out types $\mu X.\, T$ in which the equation $X = T$ has more than one solution, such as $\mu X.\, X$ or $\mu X.\, (\mu Y.\, X)$. The *value types* are the types that can be ascribed to values.

With recursive types, an issue that deserves care is the meaning of type equality. We take recursive types as abbreviations for possibly infinite trees; therefore two types T_1 and T_2 are equal, written $T_1 \equiv T_2$, if their underlying trees are the same. As shown by Amadio and Cardelli [4, section 5], relation $\equiv$ is captured by the two rules reported in Table 3 (see the literature on type systems for more discussions). We call *functional types*, written FT, the set of types whose outermost construct, modulo type equality, is an arrow type; formally T is a functional type if $T \equiv S \to \mathcal{B}$ for some S.

For reasons of simplicity, the type system does not include more basic types, such as integers or booleans, or common type constructs such as product and union types. Admittedly, this makes the language of types rather poor. It is tedious but easy to extend the definitions and the results in the paper to richer type systems. On the other hand, we do allow recursive types because they are sometimes difficult to handle: as discussed in Section 8, previous presentations of Higher-Order π-calculi and of their compilation into π-calculus forbid recursive types precisely because of technical problems in proofs. Some basic type is needed because otherwise all types would be equal to $\mu X.\, (X \to \mathcal{B})$, modulo type equality.

Channels have functional types. A channel of type $T \to \mathcal{B}$ may only carry values of type T. Therefore, if $a : T \to \mathcal{B}$, then in an input $a(x).\, P$ we can think of $(x).\, P$ as a function from T to processes (we study expressions such as $(x).\, P$, called *abstractions*, in Section 4).

Definition 2.2 *A* typing *is a finite assignment of value types to variables and of functional types to channels:*

$$\Gamma \quad ::= \quad \emptyset \;\Big|\; \Gamma, x : T \;\Big|\; \Gamma, a : FT \,.$$

A typing is closed *if no variables appear in* Γ.

Names appearing in a typing Γ are always taken to be pairwise distinct; this justifies an abuse of notation whereby Γ is regarded as a finite function from names to types: $\Gamma(p)$ is the type assigned to p by Γ. The ordering of assignments in Γ is ignored. We write $\Gamma, p : T$ to denote the typing that extends, or updates, Γ by mapping p to T.

The typing rules are reported in Table 3. Using E to range over values and processes, the typing judgements are of the form $\Gamma \vdash E : \mathit{TYPE}$. In the remainder, we often omit type $\mathcal{B}$ in typing judgements for processes and abbreviate $\Gamma \vdash P : \mathcal{B}$ as $\Gamma \vdash P$. Note that by rule T-Rest, all channels are created with a functional type. We sometimes write $\mathrm{L}\pi \triangleright \Gamma \vdash P, Q$ to mean that both $\mathrm{L}\pi \triangleright \Gamma \vdash P$ and $\mathrm{L}\pi \triangleright \Gamma \vdash Q$ hold.

Definition 2.3 *An $L\pi$ process P is* well-typed for Γ, *written $L\pi \triangleright \Gamma \vdash P$, if $\Gamma \vdash P$ can be inferred from the rules of Table 3. P is* well-typed *if there is Γ such that P is well-typed for Γ.*

Value $\star$ can only be ascribed the type `unit`. Indeed, $\star$ is the only closed value of type `unit`. Therefore if $\Gamma, a : \mathtt{unit} \to \mathcal{B} \vdash a\langle v\rangle$ and v is closed, then by rule T-Out, it must be $v = \star$.

Example 2.4 *Process $b(y).\, y\langle\star\rangle \mid a\langle b\rangle \mid a(x).\, x\langle c\rangle$ is well-typed for*

$$c : \mathtt{unit} \to \mathcal{B}, b : (\mathtt{unit} \to \mathcal{B}) \to \mathcal{B}, a : ((\mathtt{unit} \to \mathcal{B}) \to \mathcal{B}) \to \mathcal{B} \,.$$

We have, omitting $\mathbf{0}$ processes, $P \Longrightarrow c\langle\star\rangle$.

Example 2.5 *Having recursive types, we can type self-applications such as $v\langle v\rangle$. Take*

$$T \stackrel{def}{=} \mu X.\, (X \to \mathcal{B}) \tag{1}$$

Then $T \equiv T \to \mathcal{B}$; therefore $a : T \vdash a\langle a\rangle$, and also $a : T \vdash a\langle a\rangle \mid a(x).\, x\langle x\rangle$. We have $a\langle a\rangle \mid a(x).\, x\langle x\rangle \xrightarrow{\tau} \mathbf{0} \mid a\langle a\rangle$.

Example 2.6 *Process*

$$P \stackrel{def}{=} a\langle\star\rangle \mid a(x).\, x\langle v\rangle$$

cannot be typed. This can be easily established with the Subject Reduction Theorem 2.11. P should not indeed be typable, as $P \xrightarrow{\tau} \mathbf{0} \mid \star\langle v\rangle$ and the latter is a "wrong" process, as it uses the unit value as a channel.

If P is well-typed for Γ, then all free names in P are nominated in Γ, therefore:

Process typing:

$$\text{T-NIL}\ \frac{}{\Gamma \vdash \mathbf{0}\ : \mathcal{B}}$$

$$\text{T-PAR}\ \frac{\Gamma \vdash P\ : \mathcal{B} \qquad \Gamma \vdash Q\ : \mathcal{B}}{\Gamma \vdash P \mid Q\ : \mathcal{B}}$$

$$\text{T-REPL}\ \frac{\Gamma \vdash P\ : \mathcal{B}}{\Gamma \vdash !P\ : \mathcal{B}}$$

$$\text{T-REST}\ \frac{\Gamma, a : T \vdash P\ : \mathcal{B} \qquad T \in FT}{\Gamma \vdash (\boldsymbol{\nu} a : T)\, P\ : \mathcal{B}}$$

$$\text{T-IN}\ \frac{\Gamma \vdash a\ : T \to \mathcal{B} \qquad \Gamma, x : T \vdash P\ : \mathcal{B}}{\Gamma \vdash a(x).\, P\ : \mathcal{B}}$$

$$\text{T-OUT}\ \frac{\Gamma \vdash v\ : T \to \mathcal{B} \qquad \Gamma \vdash w\ : T}{\Gamma \vdash v\langle w \rangle\ : \mathcal{B}}$$

Value typing:

$$\text{TV-UNIT}\ \frac{}{\Gamma \vdash \star\ : \texttt{unit}}$$

$$\text{TV-BASE}\ \frac{\Gamma(p) = T \qquad T \equiv S}{\Gamma \vdash p\ : S}$$

where relation $\equiv$ is the least congruence on types that satisfies these two rules::

$$\frac{}{\mu X.\, T \equiv T\{{}^{\mu X.\, T}\!/X\}}$$

$$\frac{X \text{ guarded in } T \qquad T\{{}^{S_1}\!/X\} \equiv S_1 \qquad T\{{}^{S_2}\!/X\} \equiv S_2}{S_1 \equiv S_2}$$

Table 3: The typing rules for Lπ

Lemma 2.7 *If Γ is closed and $\Gamma \vdash P$, then P is also closed.*

Lemma 2.8 (weakening) *If $\Gamma \vdash P$ and $p \notin \mathrm{fn}(P)$, then for all S, we have $\Gamma, p : S \vdash P$.*

Lemma 2.9 (strengthening) *If $\Gamma, p : S \vdash P$ and p is not free in P, then $\Gamma \vdash P$.*

Lemma 2.10 (substitution) *Suppose $\Gamma \vdash P$, $\Gamma \vdash p : T$, and $\Gamma \vdash v : T$. Then $\Gamma \vdash P\{v/p\}$.*

Theorem 2.11 (subject reduction) *If $\Gamma \vdash P$, with Γ closed, and $P \xrightarrow{\tau} P'$ then $\Gamma \vdash P'$.*

Proof: By induction on the depth of the proof of $P \xrightarrow{\tau} P'$. To handle the case of rule Com one has to extend subject reduction to input and output actions. For instance, the assertion for output says that if $\Gamma \vdash P$, with Γ closed, and $P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\ \overline{a}\langle v\rangle} P'$, then there is T such that:

- $\Gamma \vdash a : T \to \mathcal{B}$;
- $\Gamma, \widetilde{b} : \widetilde{T} \vdash v : T$;
- $\Gamma, \widetilde{b} : \widetilde{T} \vdash P'$.

■

Subject reduction shows that a well-typed process can only evolve into well-typed process derivatives. This excludes run-time errors like the unit value appearing as the subject of a prefix. For instance, we can use subject reduction to prove that process P of Example 2.6 is not typable: if there were Γ such that $\Gamma \vdash P$, then also $\Gamma \vdash \mathbf{0} \mid \overline{\star}\langle v\rangle$, which is impossible because, by rule T-Out, typing $\overline{\star}\langle v\rangle$ requires $\star$ having a functional type.

2.5 The syntactic constraints of Lπ

The asynchronous π-calculus [17, 7] has no output prefix and choice (and sometimes also no matching); in addition, Lπ only allows communication of the output capability of channels and, consequently, forbids matching. We discuss these constraints below. We begin examining the meaning of asynchrony.

2.5.1 Asynchrony

In the concurrency literature, the term *asynchronous message passing* indicates a communication mechanism in which the acts of sending and of receiving a message are disconnected. Sending a message usually means dropping it in a communication medium (for instance a queue or a bag); receiving a message means retrieving it from the communication medium. Thus a message may be received arbitrarily long after is has been sent. If the communication medium has unbounded capacity, the sending of a message does not block any processes.

Lπ does not have explicit separate events for sending and receiving messages (the calculus has output and input actions but these are intended to synchronise in a single event, the communication). Also, the syntax of the calculus does not describe a communication medium. So in what sense message-passing in the asynchronous π-calculus and in Lπ is *asynchronous*? It is asynchronous because consuming a message has no effect on the process that created that message. This holds because these calculi, in contrast with the ordinary π-calculus, have no output prefixing and choice. In an output prefix $a\langle b\rangle. P$, process P is blocked until message $a\langle b\rangle$ has been consumed. In a choice $a\langle b\rangle + c(x). P$ the consumption of $a\langle b\rangle$ has the effect of discharging the summands $c(x). P$.

In asynchronous π-calculi, a message is sent simply when it is unguarded, i.e., it is not underneath an input prefix. The messages sent are scattered around and are not grouped in a communication medium. But the effect is the same as having an unbounded bag as communication medium. It is unbounded because arbitrarily many messages may have been sent and not yet consumed; it is a bag because the order in which messages are consumed is not related to the order in which the messages have been emitted. In asynchronous π-calculi, a communication $a\langle b\rangle \mid a(x). P \xrightarrow{\tau} \mathbf{0} \mid P\{b/x\}$ bears two roles: the message $a\langle b\rangle$ is consumed and therefore removed from the communication medium; the unguarded output messages of $P\{b/x\}$ are liberated and therefore added to the communication medium. Note that the coincidence between "message sent" and "message unguarded" would break if the calculus had a choice operator $P_1 + P_2$. In a process $a\langle b\rangle + c(x). P$, message $a\langle b\rangle$, although unguarded, is not really sent because its offer can vanish at any moment. And it is not the communication medium that decides if and when the offer will vanish, as it would be in an asynchronous model with a lossy communication medium. The availability of $a\langle b\rangle$ only depends on whether the other summand $c(x). P$ can perform its input at c. An asynchronous process that non-deterministically can chose to send $a\langle b\rangle$ or to receive at $c(x). P$ should be written $\tau. a\langle b\rangle + c(x). P$. Pierce and Nestmann [27] have showed that this form of choice, in which each summand is guarded by a τ or by an input prefix, can be encoded in the asynchronous π-calculus. Their encoding can also be written in Lπ.

2.5.2 Output capability

One of the most important application areas for the π-calculus is object-oriented languages. The reason is that naming is a central notion both for these languages and for the π-calculus. Objects refer to each other using names and, during computation, object acquaintances may change and new objects may be created (this is very evident in the case of imperative or concurrent object-oriented languages).

A consequence of the output capability constraint of Lπ, in a process $\boldsymbol{\nu} a\, P$ all possible inputs at a appear—and are syntactically visible—in P; no further input may be created, inside or outside P. This property is useful when Lπ is used for giving the semantics to, and reasoning about, concurrent or distributed object-oriented languages [21]. For instance, the property can guarantee unique identity of objects—a fundamental feature of objects. In an object world, the name a of an object may be transmitted; the recipient may use a to access its methods, but he/she cannot create a new object called a. When representing objects in the π-calculus, this usually translates into the constraint that the recipient of a may only use it in output. Indeed, Lπ may also be seen as a simple calculus of objects: a restriction $\boldsymbol{\nu} a\, P$ declares a new object called a.

The locality property of channels—the recipients of a channel are local to the process that has created the channel—gives the name "Local" to Lπ.

2.5.3 Matching

The absence of the *matching* construct (or similar constructs like mismatching) for testing equality between channels may be viewed as a consequence of the output capability constraint on communications, since testing the identity of a channel is a different capability from, and not implied by, the capability of using a channel in output. Matching, like testing equality between pointers in imperative languages, prevents many useful program optimisations and transformations. Also from a programming point of view the usefulness of matching is questionable.

2.5.4 Consequences of the constraints on the theory

The syntactic constraints that distinguish Lπ from the ordinary π-calculus are also motivated by the theory. In the asynchronous π-calculus without matching, the most well-known form of labeled bisimilarity, called *early bisimilarity*, is a congruence relation [16], and coincides with various other forms of bisimilarity that have been proposed for the π-calculus [33]. Among these is *ground bisimilarity*, whose input clause has no universal quantifications on the values received—it suffices that the observer supplies a single *fresh* value—and is therefore simpler to use in proofs than the other bisimilarities (see Section 3.2). For these results to hold, the absence of sum, matching, and output prefixing are all necessary. The constraint on output capability gives us, in addition, some very important algebraic laws, for instance laws for copying resources and laws for tail-call-like optimisations presented in Section 3.3.

A price to be paid for these constraints is (for Lπ as well as the asynchronous π-calculus) the loss of the axiomatisations of the ordinary π-calculus [29].

2.5.5 Consequences of the constraints on the expressiveness

Various encodings have been given that show that the loss of expressiveness moving from π-calculus to the asynchronous π-calculus and then to Lπ is limited [27, 5, 17, 7] and in any case acceptable in a model of asynchronous communications. Palamidessi [28] has shown that the general choice operator of the π-calculus cannot be encoded in an asynchronous π-calculus, by proving that the symmetric leader election problem cannot be solved in a π-calculus without choice. Such a weakness is acceptable—and somehow inevitable–in asynchronous calculi: results in the field of distributed computing such as [11] show that problems of consensus among a collection of asynchronous processes may have no solutions. As mentioned before, input-guarded choice can however be encoded in Lπ, and this is usually enough for programming purposes.

Experimental programming languages (or proposal of programming languages) such as Join [12], and Blue [8] have the syntactic constraints discussed in this section. Pict [30] has all the constraints except that on output capability. The type system of Pict makes, however, the separation between input and output capability on channels and, as we understand, the output capability is by large the most used.

3 Behavioural equivalences

3.1 Barbed congruence

We define behavioural equality using the notion of barbed congruence [26]. This is a bisimulation-based equivalence that can be defined uniformly on different calculi. Barbed congruence can be defined on any calculus possessing an interaction relation (the τ steps of the π-calculus) and an observability predicate $\downarrow_a$, for each channel a, which detects the possibility of a process of accepting a communication at a with the external environment. In the π-calculus, $P \downarrow_a$ holds if there is a derivative P' and an input or output action μ with subject a such that $P \xrightarrow{\mu} P'$.

Barbed congruence is the congruence induced by barbed bisimulation. The latter equates processes that can match each other's interactions and, at each step, are observable on the same channels. Of course, in a typed calculus the processes being compared must obey the same typing and the contexts in which they are tested must be compatible with this typing.

Below, we define barbed congruence on a generic typed process calculus $\mathcal{L}$ that has interaction and observation relations as above, and typing judgements for processes of the form $\mathcal{L} \triangleright \Gamma \vdash P$; we assume that contexts, closed typing, etc. are defined in $\mathcal{L}$ as in Lπ. An $\mathcal{L}$ *relation* is a set of pairs of closed processes in $\mathcal{L}$. A *typed $\mathcal{L}$ relation* is a set of triples $(\Delta \; ; P \; ; Q)$ where Δ is a closed typing and $\mathcal{L} \triangleright \Delta \vdash P, Q$. If $\mathcal{R}$ is such a relation, then we write $P \; \mathcal{R}_\Delta \; Q$ if $(\Delta \; ; P \; ; Q) \in \mathcal{R}$.

Definition 3.1 (strong barbed bisimulation) *An $\mathcal{L}$ relation $\mathcal{R}$ is an $\mathcal{L}$* barbed bisimulation *if $P \; \mathcal{R} \; Q$ implies:*

1. *if $P \xrightarrow{\tau} P'$ then there is Q' such that $Q \xrightarrow{\tau} Q'$ and $P' \; \mathcal{R} \; Q'$;*
2. *if $Q \xrightarrow{\tau} Q'$ then there is P' such that $P \xrightarrow{\tau} P'$ and $P' \; \mathcal{R} \; Q'$;*
3. *for each channel a, $P \downarrow_a$ iff $Q \downarrow_a$.*

Two $\mathcal{L}$ processes P and Q are barbed bisimilar*, written $\mathcal{L} \triangleright P \; \dot{\sim} \; Q$, if $P \; \mathcal{R} \; Q$ for some $\mathcal{L}$ barbed bisimulation $\mathcal{R}$.*

A *context* is a process expression with a single occurrence of a hole $[_\Delta]$ in it. For typing contexts, we add the following rule:

$$\text{T-CONT} \; \frac{}{\Gamma', \Gamma \vdash [_\Gamma] \; : \mathcal{B}}$$

Then a (Γ/Δ)-*context* is a context C with hole $[_\Delta]$ and such that $\Gamma \vdash C \; : \mathcal{B}$. A (Γ/Δ)-context, when filled in with a process obeying typing Δ, becomes a process obeying typing Γ. The typing Γ might contain names not in Δ; the converse might be true too, because of binders in the context that embrace the hole.

Definition 3.2 (strong barbed congruence) *Let Δ be a typing, with $\mathcal{L} \triangleright \Delta \vdash P, Q$. We say that processes P, Q are* strongly barbed congruent (in $\mathcal{L}$) at Δ, *written $\mathcal{L} \triangleright \Delta \vdash P \sim Q$, if, for each closed type environment Γ and (Γ/Δ)-context C, we have $\mathcal{L} \triangleright C[P] \; \dot{\sim} \; C[Q]$.*

The weak version of the equivalences, where one abstracts away from interactions, is obtained in the standard way. *Weak barbed bisimilarity*, written $\stackrel{\bullet}{\approx}$, is defined by modifying Definition 3.1 so that the transitions $P \xrightarrow{\tau} P'$ and $Q \xrightarrow{\tau} Q'$ are replaced with $P \Longrightarrow P'$ and $Q \Longrightarrow Q'$ respectively, and the predicate $\downarrow_a$ is replaced with $\Downarrow_a \stackrel{\text{def}}{=} \Longrightarrow \downarrow_a$ (the composition of the two relations). Similarly, *weak barbed congruence*, written $\approx$, is defined by replacing $\stackrel{\bullet}{\sim}$ with $\stackrel{\bullet}{\approx}$ in Definition 3.2.

Weak barbed congruence is the relation we are most interested in. We sometimes simply call it *barbed congruence*. In the remainder, we write $\mathcal{L} \triangleright \Delta \vdash P \sim Q$ and $\mathcal{L} \triangleright \Delta \vdash P \approx Q$ without recalling that P and Q must be well-typed in Δ. When there is no ambiguity, we may omit $\mathcal{L}$ and simply write $\Delta \vdash P \sim Q$ and $\Delta \vdash P \approx Q$, or even omit both $\mathcal{L}$ and Δ and write $P \sim Q$ and $P \approx Q$.

3.2 Ground bisimilarity

The main inconvenience of barbed congruence is that it uses quantification over contexts in the definition, and this can make proofs of process equalities heavy. Therefore, it is important to find proof techniques, especially formulations of bisimulation whose definition does not use context quantification. These bisimulations should imply, or (better) coincide with, barbed congruence. For instance, in CCS barbed congruence coincides with observation congruence. In the π-calculus a similar characterisation can be given in terms of the substitution closure of *early bisimilarity* [32].

In asynchronous π-calculi without matching there is a powerful technique for barbed congruence, namely *ground bisimilarity* [33]. Below is its definition adapted to typed Lπ. We write $\mathcal{O}_\Delta(a) \equiv T$ if $\mathrm{L}\pi \triangleright \Delta \vdash a : T \to \mathcal{B}$. That is, $\mathcal{O}_\Delta(a)$ gives the type of the values that may be carried along a, according to typing Δ.

The clauses of ground bisimulation, where we separate the requirements for τ-actions, input or output of unit values, output of a channel, and input of a channel, are so formulated in order to allow an immediate comparison with the clauses of ground bisimulation in LHOπ (Definition 6.1). In Definition 3.3, the information $\mathcal{O}_\Delta(a)$ is not necessary on outputs, but it is necessary on inputs, for which we have to insure that the type of the value supplied by the observer is compatible with the type of the channel that carries the value.

Definition 3.3 (ground bisimulation) *A typed Lπ relation $\mathcal{R}$ is an* Lπ ground bisimulation *if $P \; \mathcal{R}_\Delta \; Q$ implies:*

1. *if $P \xrightarrow{\tau} P'$, then there is Q' such that $Q \Longrightarrow Q'$ and $P' \; \mathcal{R}_\Delta \; Q'$;*

2. *if $\mathcal{O}_\Delta(a) \equiv \mathtt{unit}$ and $P \xrightarrow{\mu} P'$ with $\mu = a[\star]$ or $\mu = a\langle\star\rangle$, then there is Q' such that $Q \stackrel{\mu}{\Longrightarrow} Q'$ and $P' \; \mathcal{R}_\Delta \; Q'$;*

3. *if $\mathcal{O}_\Delta(a) \equiv T \to \mathcal{B}$ and $P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\, a\langle v\rangle} P'$, with $\widetilde{b}$ fresh for Δ, then there is Q' such that $Q \stackrel{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\, a\langle v\rangle}{\Longrightarrow} Q'$ and $P' \; \mathcal{R}_{\Delta,\widetilde{b}:\widetilde{T}} \; Q'$;*

4. *if $\mathcal{O}_\Delta(a) \equiv T \to \mathcal{B}$, d is fresh for Δ, and $P \xrightarrow{a[d]} P'$, then there is Q' such that $Q \stackrel{a[d]}{\Longrightarrow} Q'$ and $P' \; \mathcal{R}_{\Delta,d:T\to\mathcal{B}} \; Q'$.*

Two $L\pi$ processes P and Q are ground bisimilar at Δ, *written* $L\pi \triangleright \Delta \vdash P \simeq Q$, *if* $P \ \mathcal{R}_\Delta \ Q$, *for some typed $L\pi$ relation $\mathcal{R}$.*

In clause (4) of Definition 3.3 it is enough to pick some *fresh* name d, because ground bisimilarity is preserved by substitutions of channels (Theorem 3.6). In other words, adding the clause

$$\begin{array}{l}\text{if } \mathcal{O}_\Delta(a) \equiv T \to \mathcal{B}\text{, then for all } d \text{ such that } \Delta \vdash d : T \text{ , if } P \xrightarrow{a[d]} P', \\ \text{then there is } Q' \text{ such that } Q \overset{a[d]}{\Longrightarrow} Q' \text{ and } P' \ \mathcal{R}_{\Delta, d:T \to \mathcal{B}} \ Q'\end{array} \tag{2}$$

would not affect the resulting ground bisimilarity relation. Similarly, the fact that ground bisimilarity is preserved by substitutions of channels, and hence by injective substitutions on channels, implies that in the same clause (4) of Definition 3.3 it is enough to pick *some* fresh name d—the specific choice of the fresh name is irrelevant.

Thus, at a functional type it suffices to test the input of processes on a *single* value. This is remarkable, for a functional type is inhabited by an infinite number of syntactically different values. The fact that the meaning of agents can be captured without considering more than one instance of input value is an important and useful property in equivalence and model checking, sometimes called *data-independence* [22]. We should stress that the data independence property of Lπ only holds on input of channels. Had we included in Lπ, for instance, integers or booleans, then there would be a universal quantification in their input clause such as (2) above.

We extend the definition of ground bisimilarity to open processes by induction on the total number of variables in the type environment (type environments are *finite* assignments of types to names), and using closing substitutions in the same way as in the input clauses.

Definition 3.4 *Suppose* $L\pi \triangleright \Gamma, x : T \vdash P, Q$. *We write* $L\pi \triangleright \Gamma, x : T \vdash P \simeq Q$ *if*

1. *either* $T \equiv \texttt{unit}$ *and* $L\pi \triangleright \Gamma \vdash P\{{}^{\star}\!/\!{}_x\} \simeq Q\{{}^{\star}\!/\!{}_x\}$,
2. *or* $T \equiv S \to \mathcal{B}$ *and* $L\pi \triangleright \Gamma, d : T \vdash P\{{}^{d}\!/\!{}_x\} \simeq Q\{{}^{d}\!/\!{}_x\}$, *for some d fresh for* Γ.

In Lπ (in general, in asynchronous π-calculi without a matching construct), ground bisimilarity is a congruence relation and implies barbed congruence. Theorems 3.5-3.7 are proved like analogous results for asynchronous π-calculi, see [33, 3, 23].

Theorem 3.5 *$L\pi$ ground bisimilarity is a congruence relation. That is, if* $L\pi \triangleright \Delta \vdash P \simeq Q$ *and C is a* $(\Gamma/\Delta)-$*context, then also* $L\pi \triangleright \Gamma \vdash C[P] \simeq C[Q]$.

Theorem 3.6 *Suppose* $\Delta(d) \equiv \Delta(a)$. *Then* $L\pi \triangleright \Delta \vdash P \simeq Q$ *implies* $L\pi \triangleright \Delta \vdash P\{{}^{d}\!/\!{}_a\} \simeq Q\{{}^{d}\!/\!{}_a\}$.

Theorem 3.7 *If* $L\pi \triangleright \Delta \vdash P \simeq Q$, *then also* $L\pi \triangleright \Delta \vdash P \approx Q$.

The converse of Theorem 3.7 does not hold (barbed congruence is coarser); a counterexample is given by law (3) below. This is due to the output capability constraint on communications in Lπ: since the recipient of a name may only use it in output position, it cannot "observe" the identity of that name. To get a characterisation of barbed congruence, the output clause of ground bisimulation has to be weakened. Details may be found in [23].

3.3 Some laws for barbed congruence in Lπ

For ease of reference, in this section we have collected some simple laws for barbed congruence in Lπ that we shall apply several times in the paper.

This Lπ law is valid for barbed congruence, but not for ground bisimilarity:

$$b\langle a\rangle \quad = \quad (\boldsymbol{\nu} c : T)\,(b\langle c\rangle \mid !c(x).\,a\langle x\rangle) \tag{3}$$

The law is false for ground bisimilarity because it equates processes that perform syntactically different outputs: the process on the left makes the output of a global name, whereas that on the right the output of a local name.

Laws like (3) are powerful. By applying variants of (3) with more sophisticated forms of replication, every Lπ process can be transformed into a process that may only emit *private* channels. On the resulting processes, ground bisimilarity and barbed congruence coincide (on image-finite processes) [23]. We therefore have another proof technique for barbed congruence on Lπ (more expensive but more powerful than that of Theorem 3.7): apply law (3) and its variants so to obtain processes that may only emit private channels, and then use ground bisimilarity. Law (3), applied from right to left, also allows tail-call-like optimisations, for instance in the representation of functions as processes [23].

Here are some basic laws for restriction and parallel composition:

Lemma 3.8 *1.* $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(P \mid Q) \sim P \mid (\boldsymbol{\nu} a : T)\,Q$*, if a is not free in P;*

2. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,P \sim P$ *, if a is not free in P;*

3. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(\boldsymbol{\nu} b : S)\,(P) \sim (\boldsymbol{\nu} b : S)\,(\boldsymbol{\nu} a : T)\,(P)$*;*

4. (abelian monoidal laws for parallel composition):

(i) : $\Gamma \vdash P \mid Q \sim Q \mid P$,
(ii) : $\Gamma \vdash P \mid (Q \mid R) \sim (P \mid Q) \mid R$,
(iii) : $\Gamma \vdash P \mid \mathbf{0} \sim P$;

5. $\Gamma \vdash !P \sim P \mid !P$;

6. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(a(x).\,P \mid a\langle v\rangle) \approx (\boldsymbol{\nu} a : T)\,(P\{^v\!/\!_x\})$.

A name *a occurs free in P only in output position* if all free occurrences of a in P are inside an output particle; for instance this holds for $P = a\langle v\rangle \mid b\langle a\rangle$. We report some distributivity laws for private replications, i.e., for systems of the form

$$(\boldsymbol{\nu} a : T)\,(P \mid !a(x).\,Q)$$

in which a may occur free in P and Q only in output position. One should think of Q as a private resource of P; indeed P can activate as many copies of Q as needed. The resource Q, however, like a recursively-defined function, when activated, can invoke itself.

In Lemma 3.9, law (1) is the distributivity law of private replication over parallel composition. Law (2) is the distributivity over replication; it can be thought of as a generalisation of law (1) to an infinite number of parallel processes. Also law (7) can be regarded as a form of distributivity for private replications, namely the distributivity over outputs: on the left-hand side the two names of the output $a\langle a\rangle$ refer to the same resource; on the right-hand side each component refers to a separate copy. Law (3-4) are commutativity laws of private replication with respect to input prefix and restriction. Law (5) is a garbage-collection law. Law (6) is a form of β-conversion: the resource located at a is applied to the argument of the output. Finally law (8) allows us to drop the replication in front of a resource that may be used at most once.

Laws (3-6) and (8) are valid also in the ordinary π-calculus. By contrast, for the distributivity laws (1), (2) and (7) the constraint that only the output capability of names may be transmitted is necessary. Indeed, the possibility of using these laws (or law (3) at the beginning of this section) is, from the theory side, one of the most attractive features of Lπ. These laws play a key role in the proofs of full abstraction in Sections 5 and 7.

Lemma 3.9 *Suppose a occurs free in P, R, Q only in output position. Then:*

1. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(P \mid R \mid\, !a(x).\,Q) \sim (\boldsymbol{\nu} a : T)\,(P \mid\, !a(x).\,Q) \mid (\boldsymbol{\nu} a : T)\,(R \mid\, !a(x).\,Q)$;
2. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,((!P) \mid\, !a(x).\,Q) \sim\, !(\boldsymbol{\nu} a : T)\,(P \mid\, !a(x).\,Q)$;
3. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(b(y).\,P \mid\, !a(x).\,Q) \sim b(y).\,(\boldsymbol{\nu} a : T)\,(P \mid\, !a(x).\,Q)$, *if y is not free in Q and $a \neq b$;*
4. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(((\boldsymbol{\nu} c : S)\,P) \mid\, !a(x).\,Q) \sim (\boldsymbol{\nu} c : S)\,(\boldsymbol{\nu} a : T)\,(P \mid\, !a(x).\,Q)$, *if c is not free in $a(x).\,Q$;*
5. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(P \mid\, !a(x).\,Q) \sim P$, *if a is not free in P;*
6. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(a\langle v\rangle \mid\, !a(x).\,Q) \approx (\boldsymbol{\nu} a : T)\,(Q\{v/x\} \mid\, !a(x).\,Q)$;
7. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(a\langle a\rangle \mid\, !a(x).\,Q) \sim (\boldsymbol{\nu} a : T, a' : T\,)(a\langle a'\rangle \mid\, !a(x).\,Q \mid\, !a'(x).\,Q)$, *if a' is not free in $a(x).\,Q$;*
8. $\Gamma \vdash (\boldsymbol{\nu} a : T)\,(P \mid\, !a(x).\,Q) \sim (\boldsymbol{\nu} a : T)\,(P \mid a(x).\,Q)$ *if a not free in Q and a appears free in P only once and in output subject position.*

Remark 3.10 The laws of Lemma 3.8 and the laws (3-6) and (8) of Lemma 3.9 are also valid for ground bisimilarity. The remaining laws (laws (1), (2), (7) of Lemma 3.9) are valid for ground bisimilarity under these conditions: for laws (1) and (2), a may only appear free in P, R and Q in output subject position; for law (7), a and x may only appear free in Q in output subject position.

4 Higher-order Lπ

In a *higher-order* process calculus, terms of the language can be passed around. Sometimes in the literature, higher-order process calculi are called *process-passing* calculi. The higher-order paradigm inherits from the λ-calculus the idea that a computation step involves instantiation of variables with terms. In this section we extend Lπ and move to higher-order, by allowing values built out of processes. We call the resulting calculus *Local Higher-Order π-calculus*, briefly LHOπ. Its syntax and operational semantics are defined by adding the productions and rules in Table 4 to those of Lπ.

Passing a process is like passing a parameterless procedure. The recipient of a process can do nothing with it but execute it, possibly several times. Procedures gain great utility if they can be *parametrised* so that, when invoked, some arguments may be supplied. In the same way a higher-order process calculus gains power if the processes that are communicated may be parametrised (see also Remark 5.13).

A parametrised process, that we call an *abstraction*, is an expression of the form $(x : T).\, P$. We may also regard abstractions as a component of input prefixes, viewing an input $a(x).\, P$ as an abstraction *located* at a. Indeed, the part $(x).\, P$ of $a(x).\, P$ behaves exactly like an abstraction; the only difference is that the bound name x of the input need not be annotated with a type, because this can be inferred from that of channel a. We omit the type annotation also in abstractions when this type is not important or it is clear from the context.

When an abstraction $(x : T).\, P$ is applied to an argument w it yields the process $P\{{}^{w}\!/\!{}_{x}\}$. Application is the destructor for abstractions. The application of v to w is written $v\langle w\rangle$. We use the same syntax of outputs because an output $a\langle w\rangle$ may be regarded as a form application, namely the application of an abstraction located at a to w. That is, we may think of $a(x).\, P \mid a\langle w\rangle \xrightarrow{\tau} P\{{}^{w}\!/\!{}_{x}\}$ as a *located interaction*, as opposed to a *non-located interaction* $((x : T).\, P)\langle w\rangle \xrightarrow{\tau} P\{{}^{w}\!/\!{}_{x}\}$. In the former case, channel a is needed in order to identify the abstraction $(x).\, P$ and the argument w, since they may not be in contiguous positions. A presentation of LHOπ in which input prefixes are constructed from abstractions, and located interactions are defined in terms of the non-located ones, is discussed in Section 4.1. Note that since both abstractions and channels may have a functional type, $b\langle a\rangle$ and $b\langle (x : T).\, P\rangle$ may be typable under the *same* type assignment (see Examples 4.3 and 4.4 below).

In LHOπ, abstractions can be communicated, but not processes themselves. To send a process, say P, we must first add a dummy parameter, for instance a parameter of unit type as in $(x : \texttt{unit}).\, P$ (for x not free in P). We call an abstraction of type $\texttt{unit} \to \mathcal{B}$ a *process value.* We forbid direct communications of processes for economy in the calculus (passing processes would require a further production in the grammar of values and of processes) and because it does not add expressiveness. Note also that, due to the separation between values and processes, in LHOπ a process can execute iff it is not inside an abstraction (calling abstraction also the subexpression $(x).\, P$ of an input $a(x).\, P$).

Example 4.1 *Here are a process Q that is willing to send a process P along a channel a, and a process R that is willing to receive and execute what Q is* *sending on a:*

$$\begin{array}{rcl} Q & \stackrel{def}{=} & a\langle (x:\texttt{unit}).\,P\rangle \\ R & \stackrel{def}{=} & a(y).\,y\langle\star\rangle\ . \end{array}$$

These processes interact as follows:

$$\begin{array}{rcl} Q \mid R & \xrightarrow{\tau} & ((x:\texttt{unit}).\,P)\langle\star\rangle \\ & \xrightarrow{\tau} & P\{{}^{\star}\!/\!x\}\ . \end{array}$$

Example 4.2 *$v \stackrel{def}{=} (z:\texttt{unit}\to\mathcal{B}).\,(P \mid z\langle\star\rangle)$ is an abstraction of type $(\texttt{unit}\to\mathcal{B})\to\mathcal{B}$, and represents a function from process values to processes, which runs the process-argument in parallel with P. We have:*

$$v\langle (x:\texttt{unit}).\,R\rangle \xrightarrow{\tau} P \mid ((x:\texttt{unit}).\,R)\langle\star\rangle \xrightarrow{\tau} P \mid R\{{}^{\star}\!/\!x\}\ .$$

$w \stackrel{def}{=} (y:(\texttt{unit}\to\mathcal{B})\to\mathcal{B}).\,(P \mid y\langle((x:\texttt{unit}).\,R)\rangle)$ has type $((\texttt{unit}\to\mathcal{B})\to\mathcal{B})\to\mathcal{B}$, and takes abstractions of the same type as v as argument. We have:

$$w\langle v\rangle \Longrightarrow P \mid P \mid R\{{}^{\star}\!/\!x\}.$$

The two examples below show that a channel can harmlessly emit both channels and abstractions. We omit type annotations and we garbage-collect **0** processes that are produced in the reductions.

Example 4.3 *For all S, the process*

$$P \stackrel{def}{=} a\langle d\rangle \mid a\langle (y).\,d\langle y\rangle\rangle \mid a(x).\,x\langle c\rangle$$

is well-typed for $c:S, d:S\to\mathcal{B}, a:(S\to\mathcal{B})\to\mathcal{B}$. The process has two reduction sequences, both of which generate the message $d\langle c\rangle$:

$$\begin{array}{rclcl} P & \xrightarrow{\tau} & a\langle (y).\,d\langle y\rangle\rangle \mid d\langle c\rangle & & \\ P & \xrightarrow{\tau} & a\langle d\rangle \mid ((y).\,d\langle y\rangle)\langle c\rangle & \xrightarrow{\tau} & a\langle d\rangle \mid d\langle c\rangle\ . \end{array}$$

Example 4.4 *Process*

$$P \stackrel{def}{=} a\langle b\rangle \mid a(x).\,x\langle c\rangle \mid a\langle (y).\,y\langle e\rangle\rangle \mid b(y).\,y\langle d\rangle$$

is well-typed for

$$d:S, e:S, c:S\to\mathcal{B}, b:(S\to\mathcal{B})\to\mathcal{B}, a:((S\to\mathcal{B})\to\mathcal{B})\to\mathcal{B}$$

for all S. The process has two possible initial reductions, depending on which output at a is chosen. In one case the value sent at a is a channel, in the other case an abstraction:

$$\begin{array}{rcl} P & \xrightarrow{\tau} & b\langle c\rangle \mid a\langle (y).\,y\langle e\rangle\rangle \mid b(y).\,y\langle d\rangle \\ & \xrightarrow{\tau} & a\langle (y).\,y\langle e\rangle\rangle \mid c\langle d\rangle \\ P & \xrightarrow{\tau} & a\langle b\rangle \mid ((y).\,y\langle e\rangle)\langle c\rangle \mid b(y).\,y\langle d\rangle \\ & \xrightarrow{\tau} & a\langle b\rangle \mid c\langle e\rangle \mid b(y).\,y\langle d\rangle\ . \end{array}$$

The fact that a variable may be instantiated both with a channel and with an abstraction, that is the fact that abstractions and channels may be given the same type, is possible because of all syntactic constraints of Lπ (and hence also of LHOπ) discussed in Section 2.5. For instance, if we had output prefixing like in the ordinary π-calculus, then replacing x in $x\langle v\rangle. P$ with an abstraction, say $(y). Q$, would yield an expression that reduces to $Q\{v/y\}. P$, which is not a process expression because it contains a sequential composition between processes.

The types of values are unique, up to type equality:

Lemma 4.5 *If LHO$\pi \triangleright \Gamma \vdash v : T$ and LHO$\pi \triangleright \Gamma \vdash v : S$ then $T \equiv S$.*

Remark 4.6 The values of LHOπ cannot autonomously reduce. We might say that they are in normal form. This property would not hold if, for instance, the grammar of values allowed applications and nesting of abstractions. In this case, $v \stackrel{\text{def}}{=} ((x).(y).P)\langle w\rangle$ would be a value, and $a\langle v\rangle$ would be a legal process expression. However v is not a normal form, for it has a reduction $v \xrightarrow{\tau} (y). P\{w/x\}$, the result of which is a value in normal form. If values were other than normal forms, then we would have to specify a reduction strategy for them, in order to say, for instance, if in expressions such as $a\langle v\rangle \mid a(x). P$, the value v should be reduced to a normal form before the communication at a takes place (a positive answer would mean adopting a call-by-value strategy). We require that values in LHOπ be normal forms in order to avoid specifying a reduction strategy for value expressions and, instead, concentrate on basic issues of higher-order process calculi, including their expressivity. We regard reduction strategies as an important but orthogonal issue. Reduction strategies are best understood separately, on the λ-calculus. Encodings of various λ-calculus reduction strategies into π-calculus are studied in [25, 37]. ■

Barbed congruence is defined in LHOπ as by Definition 3.2 (take the calculus $\mathcal{L}$ in the definition to be LHOπ).

Remark 4.7 The language of LHOπ without replication is enough to write processes with an infinite behaviour, for the same reason why the paradoxical operator **Y** can be written in the λ-calculus. We incorporate replication in the syntax because it will facilitate the comparison with the first-order calculus Lπ, whose operators include replication. A process $!P$ can be written without the outermost replication as follows. As seen in (1), type $T \stackrel{\text{def}}{=} \mu X.(X \to \mathcal{B})$ can be used to type self-application. Let $v_P \stackrel{\text{def}}{=} (x : T).(P \mid x\langle x\rangle)$, for x not free in P. If $\Gamma \vdash P$, then $\Gamma \vdash v_P : T$. Writing P^n for n copies of P in parallel, we have $v_P\langle v_P\rangle \Longrightarrow P^n \mid v_P\langle v_P\rangle$, for all n; indeed it holds that LHO$\pi \triangleright \Gamma \vdash v_P\langle v_P\rangle \approx !P$. ■

Since LHOπ values may contain processes, it is useful to define barbed congruence also on values. We call a $(\Gamma/\Delta /\!\!/ T)$-context a context that, when filled in with a value v such that LHO$\pi \triangleright \Delta \vdash v : T$, becomes a process obeying typing Γ. Formally, we call a *value context* a process expression in which an occurrence of a value has been replaced by a hole of the form $[_{\Gamma;T}]$. We add the typing rule

$$\textsc{Tv-Cont}\ \frac{}{\Gamma', \Gamma \vdash [_{\Gamma;T}] : T}$$

Productions and rules to be added to those for Lπ (Table 1-3):

Grammar for values:

$$v \quad ::= \quad (x:T).\,P \quad \text{abstraction}$$

Operational rules:

$$\text{R-APP}\ \frac{}{((x:T).\,P)\langle v\rangle \xrightarrow{\tau} P\{v/x\}}$$

Typing rules for values:

$$\texttt{Tv-abs}\frac{\Gamma, x:T \vdash P}{\Gamma \vdash (x:T).\,P \ : T \to \mathcal{B}}$$

Table 4: The calculus LHOπ

---

and say that a $(\Gamma/\Delta /\!\!/ T)$-*context* is a value context C with hole $[_{\Delta;T}]$ such that $\Gamma \vdash C\ :\mathcal{B}$.

Definition 4.8 (barbed congruence on values) *Suppose LHOπ $\triangleright \Delta \vdash v_i\ :\ T\quad (i=1,2)$. We say that v_1 and v_2 are* barbed congruent at $(\Delta;T)$, *written LHOπ $\triangleright \Delta \vdash v_1 \approx v_2 : T$ if, for each closed type environment Γ and $(\Gamma/\Delta/\!\!/T)$-context C, we have LHOπ $\triangleright C[v_1]\ \ \dot{\approx}\ \ C[v_2]$.*

For manipulating values, the following lemma is useful; it relates barbed congruence between two values to barbed congruence between two processes.

Lemma 4.9 *Suppose LHOπ $\triangleright\ \Delta \vdash v_i\ :\ T\quad (i=1,2)$, and a is fresh for Δ. Then LHOπ $\triangleright \Delta \vdash v_1 \approx v_2 : T$ iff LHOπ $\triangleright \Delta, a : T \to \mathcal{B} \vdash a\langle v_1\rangle \approx a\langle v_2\rangle$.*

Proof: The implication from left to right is true by the definition of barbed congruence on processes (Definition 3.2) and on values (Definition 4.8): both definitions use quantification on contexts, and a context for the $a\langle v_i\rangle$'s is also a context for the v_i's. For the opposite implication we have to show that for each closed type environment Γ and $(\Gamma/\Delta/\!\!/T)$-context C, we have LHOπ $\triangleright C[v_1]\ \ \dot{\approx}\ \ C[v_2]$. We prove, by induction on C, that we can find a $(\Gamma/\Delta, a : T \to \mathcal{B}/\!\!/T)$-context D such that $C[v_i]\ \ \dot{\approx}\ \ D[a\langle v_i\rangle]$, for some a fresh for v_i $(i = 1, 2)$. This would conclude the proof because from the hypothesis LHOπ $\triangleright \Delta, a : T \to \mathcal{B} \vdash a\langle v_1\rangle \approx a\langle v_2\rangle$ we have

$$D[a\langle v_1\rangle]\ \ \dot{\approx}\ \ D[a\langle v_2\rangle],$$

hence also $C[v_1]\ \ \dot{\approx}\ \ C[v_2]$, by transitivity.

We only consider the base case when $C \stackrel{\mathrm{def}}{=} w\langle[\cdot]\rangle$. Define (we omit types) $D \stackrel{\mathrm{def}}{=} \boldsymbol{\nu} a\,([\cdot] \mid a(x).\, w\langle x\rangle)$, for some a fresh. We have

$$\begin{array}{rcl} D[a\langle v_i\rangle] & = & \boldsymbol{\nu} a\,(a\langle v_i\rangle \mid a(x).\, w\langle x\rangle) \\ & \approx & w\langle v_i\rangle \\ & = & C[v_i] \end{array}$$

■

4.1 A functional presentation of LHOπ

We outline a different formulation of LHOπ in which functional values, that is values of functional type, have an even more prominent role.

An input $a(x).\,P$ has a channel part a and a functional-value part $(x).\,P$ (we assume here that abstractions have no type annotations); accordingly the syntax of input prefix can be $p\,v$ (a similar construct is used in the Blue calculus [8]). Thus if $a : T \to \mathcal{B}$ and $b : T$ then $a\,b$ is a legal process expression (indeed, $a\,b$ can be η-expanded to $a(x).\,b\langle x\rangle$). The type system would rule out expressions $p\,v$ in which v has unit type. Functional values can also be used as the targets of input and output transitions; input transitions could be of the form $P \xrightarrow{a} v$, and output transitions of the form $P \xrightarrow{\overline{a}} v$. For instance, we would have:

$$\begin{array}{ccc} a\,v \xrightarrow{a} v & & a\langle v\rangle \xrightarrow{\overline{a}} (x).\, x\langle v\rangle \\ & \boldsymbol{\nu} b\, a\langle b\rangle \xrightarrow{\overline{a}} (x).\, \boldsymbol{\nu} b\, x\langle b\rangle & \end{array}$$

In an output $P \xrightarrow{\overline{a}} (x).\,P'$, variable x represents the recipient of the value transmitted in the output. The communication rule would then be

$$\frac{P_1 \xrightarrow{\overline{a}} v_1 \quad P_2 \xrightarrow{a} v_2}{P_1 \mid P_2 \xrightarrow{\tau} v_1\langle v_2\rangle}$$

so that the only rule involving substitutions would be R-App. Examples:

$$\begin{array}{rcl} a\langle v\rangle \mid a(y).\,P & \xrightarrow{\tau} & ((x).\,x\langle v\rangle)\langle (y).\,P\rangle \\ & \xrightarrow{\tau} & ((y).\,P)\langle v\rangle \\ & \xrightarrow{\tau} & P\{v/y\} \end{array}$$

$$\begin{array}{rcl} a\,d \mid a\langle v\rangle \mid d(y).\,P & \xrightarrow{\tau} & ((x).\,x\langle v\rangle)\langle d\rangle \mid d(y).\,P \\ & \xrightarrow{\tau} & d\langle v\rangle \mid d(y).\,P \\ & \xrightarrow{\tau} & ((x).\,x\langle v\rangle)\langle (y).\,P\rangle \\ & \xrightarrow{\tau} & ((y).\,P)\langle v\rangle \\ & \xrightarrow{\tau} & P\{v/y\}\ . \end{array}$$

We shall come back to this formulation when we discuss labeled bisimilarity in LHOπ, in Remark 6.2 of Section 6. A similar, functional, formulation can be given of Lπ, as well as of other π-calculus-like languages [9].

5 Compiling higher-order into first-order

We show that the higher-order calculus LHOπ can be faithfully compiled down into the first-order calculus Lπ. This shows that the higher-order features of

Translation of higher-order values:

$$\begin{array}{rcll}
[\![(x:T).\,P]\!]_a^\Gamma & \stackrel{\text{def}}{=} & a(x).\,[\![P]\!]^{\Gamma,x:T} & \\
[\![q]\!]_a^\Gamma & \stackrel{\text{def}}{=} & a(y).\,[\![q\langle y\rangle]\!]^{\Gamma,y:S} & \text{if } \Gamma \vdash q : S \to \mathcal{B}
\end{array}$$

Translation of processes:

$$[\![v\langle w\rangle]\!]^\Gamma \stackrel{\text{def}}{=} \begin{cases}
v\langle w\rangle & \text{if } v \text{ is a name and } S \equiv \mathtt{unit} \\
(\boldsymbol{\nu} a : \mathtt{unit} \to \mathcal{B})\,([\![v]\!]_a^\Gamma \mid a\langle w\rangle) & \\
\qquad\qquad \text{if } v \text{ is not a name and } S \equiv \mathtt{unit} & \\
(\boldsymbol{\nu} a : S)\,(v\langle a\rangle \mid ![\![w]\!]_a^\Gamma) & \\
\qquad\qquad \text{if } v \text{ is a name and } S \text{ is a functional type} & \\
(\boldsymbol{\nu} a : S \to \mathcal{B}, b : S\,)([\![v]\!]_a^\Gamma \mid a\langle b\rangle \mid ![\![w]\!]_b^\Gamma) & \\
\qquad\qquad \text{if } v \text{ is not a name and } S \text{ is a functional type} &
\end{cases}$$

where a, b are fresh for v, w and $\Gamma \vdash w : S$.

$$\begin{array}{rcl}
[\![P \mid Q]\!]^\Gamma & \stackrel{\text{def}}{=} & [\![P]\!]^\Gamma \mid [\![Q]\!]^\Gamma \\
[\![a(x).\,P]\!]^\Gamma & \stackrel{\text{def}}{=} & a(x).\,[\![P]\!]^{\Gamma,x:T} \quad \text{if } \Gamma \vdash a : T \to \mathcal{B} \\
[\![(\boldsymbol{\nu} a : T)\,P]\!]^\Gamma & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} a : T)\,[\![P]\!]^{\Gamma,a:T} \\
[\![!P]\!]^\Gamma & \stackrel{\text{def}}{=} & ![\![P]\!]^\Gamma \\
[\![\mathbf{0}]\!]^\Gamma & \stackrel{\text{def}}{=} & \mathbf{0}
\end{array}$$

Table 5: The compilation $[\![\,.\,]\!]$ of LHOπ^- into Lπ^-

LHOπ, although convenient from a programming point of view, do not add expressive power. The compilation also allows us to exploit the theory of first-order calculi to reason about higher-order calculi (see Sections 5.1 and 6).

We first present a compilation that has nice properties and whose definition and full abstraction can be extended to calculi without the syntactic constraints of Lπ, including synchronous calculi. It is, essentially, the compilation in [32, 35]. The compilation however is not defined on recursive types; we will consider these in Section 7. We call LHOπ^- and Lπ^- the subcalculi of LHOπ and Lπ, respectively, without recursive types.

Table 5 defines the compilation from LHOπ^- to Lπ^- on processes and higher-order values. The communication of an abstraction v is translated as the communication of a private name that acts as a pointer to (the translation of) v and that the recipient can use to trigger copies of (the translation of) v with appropriate arguments. For instance a process $a\langle (x).\,R\rangle$ is translated into the process $\boldsymbol{\nu} b\,(a\langle b\rangle \mid !b(x).\,[\![R]\!])$; a recipient of the pointer b can use it to activate as many copies of $[\![R]\!]$ as needed. In the translation of an application $v\langle w\rangle$ where v is an abstraction, the function v is located at some fresh name a; the function is activated by receiving an argument at a. This argument is w itself, if w has unit type; it is a trigger for w if w has a functional type.

The compilation is the identity on types and type environments. Therefore the compilation does not modify the types of names, and an abstraction $(x : T).\, P$ is translated into a process $a(x).\, P'$ where a has type $T \to \mathcal{B}$. The translation of processes and values is annotated with a type environment Γ as parameter. This is needed for putting the necessary type annotations in the names introduced by the compilation. The translation of a name q uses $[\![q\langle y\rangle]\!]^{\Gamma, y:S}$ that, if y has a functional type, requires the translation of name y. This definition is well-founded (that is, $[\![\,.\,]\!]$ terminates) because the order of the type of y is smaller than that of the type of q. The translation would loop if the types were recursive, which explains why recursive types are forbidden. In the examples below, we omit the obvious type annotations and we garbage-collect $\mathbf{0}$ processes and restrictions whenever possible.

Example 5.1 *Let R be any closed process, and $P \stackrel{def}{=} ((z).\, z\langle\star\rangle)\langle(x).\, R\rangle$. It holds that $P \xrightarrow{\tau} ((x).\, R)\langle\star\rangle \xrightarrow{\tau} R\{\star/x\}$. The translation of P is (for a, b fresh)*

$$[\![P]\!] = (\boldsymbol{\nu} a, b\,)\Big(a(z).\, z\langle\star\rangle \mid a\langle b\rangle \mid !b(x).\, [\![R]\!]\Big)$$

and we have:

$$\begin{array}{rcll} [\![P]\!] & \xrightarrow{\tau} & \boldsymbol{\nu} b\, (b\langle\star\rangle \mid !b(x).\, [\![R]\!]) & \\ & \xrightarrow{\tau} & \boldsymbol{\nu} b\, ([\![R]\!]\{\star/x\} \mid !b(x).\, [\![R]\!]) & \\ & \sim & [\![R\{\star/x\}]\!] & (4) \end{array}$$

where (4) is derived from law (5) of Lemma 3.9.

Example 5.2 *Let $v \stackrel{def}{=} (z).\, (z\langle\star\rangle \mid z\langle\star\rangle)$, and $P \stackrel{def}{=} a\langle v\rangle \mid a(y).\, y\langle(x).\, R\rangle$. We have $P \Longrightarrow R\{\star/x\} \mid R\{\star/x\}$. The translation of P is*

$$[\![P]\!] = \boldsymbol{\nu} b\, (a\langle b\rangle \mid !b(z).\, (z\langle\star\rangle \mid z\langle\star\rangle)) \mid a(y).\, \boldsymbol{\nu} c\, (y\langle c\rangle \mid !c(x).\, [\![R]\!])$$

and we have, proceeding as in the previous example:

$$\begin{array}{rcl} [\![P]\!] & \xrightarrow{\tau} & \boldsymbol{\nu} b\, (!b(z).\, (z\langle\star\rangle \mid z\langle\star\rangle)) \mid \boldsymbol{\nu} c\, (b\langle c\rangle \mid !c(x).\, [\![R]\!]) \\ & \xrightarrow{\tau} & (\boldsymbol{\nu} b, c\,)(!b(z).\, (z\langle\star\rangle \mid z\langle\star\rangle) \mid c\langle\star\rangle \mid c\langle\star\rangle \mid !c(x).\, [\![R]\!]) \\ & \xrightarrow{\tau}\xrightarrow{\tau} & (\boldsymbol{\nu} b, c\,)(!b(z).\, (z\langle\star\rangle \mid z\langle\star\rangle) \mid !c(x).\, [\![R]\!] \mid [\![R]\!]\{\star/x\} \mid [\![R]\!]\{\star/x\}) \\ & \sim & [\![R\{\star/x\}]\!] \mid [\![R\{\star/x\}]\!] \\ & = & [\![R\{\star/x\} \mid R\{\star/x\}]\!]. \end{array}$$

The computation of a process $[\![P]\!]$ may require more steps (i.e., more interactions) than the corresponding computation by P. But if we do not weight internal work, then P and $[\![P]\!]$ have the "same" behaviour.

Proposition 5.3 *For all process P, value v, and functional type T:*

1. *$LHO\pi^- \triangleright \Gamma \vdash P$ implies $L\pi^- \triangleright \Gamma \vdash [\![P]\!]^\Gamma$;*

2. $LHO\pi^- \triangleright \Gamma \vdash v : T$ *implies* $L\pi^- \triangleright \Gamma, p : T \vdash [\![v]\!]_p^\Gamma$.

We prove that there is a precise operational correspondence between actions of P and of its transformed $[\![P]\!]^\Gamma$. In the assertion of the results, in particular Theorems 5.8 and 5.9, we use ground bisimilarity; we recall that it implies barbed congruence. We first give the result for visible actions. Lemmas 5.4-5.6 are proved using induction on the depth of the proof of $P \xrightarrow{\mu} P'$ (item 1) and of $[\![P]\!]^\Gamma \xrightarrow{\mu} P''$ (item 2), and using the laws of Lemma 3.8. We recall that $\mathcal{O}_\Delta(a)$ is the type of the values that may be carried along a.

Lemma 5.4 (operational correspondence of $[\![\,.\,]\!]$ on synchronisations) *Suppose* $LHO\pi^- \triangleright \Gamma \vdash P$, *with* Γ *closed,* $\mathcal{O}_\Gamma(a) \equiv \mathtt{unit}$, *and* μ *is an input* $a[\star]$ *or an output* $a\langle\star\rangle$.

1. *If* $P \xrightarrow{\mu} P'$ *then* $[\![P]\!]^\Gamma \xrightarrow{\mu} [\![P']\!]^\Gamma$;
2. *the converse, i.e., if* $[\![P]\!]^\Gamma \xrightarrow{\mu} P''$, *then there is* P' *such that* $P \xrightarrow{\mu} P'$ *and* $P'' = [\![P']\!]^\Gamma$.

Lemma 5.5 (operational correspondence of $[\![\,.\,]\!]$ on higher-order input actions) *Suppose* $LHO\pi^- \triangleright \Gamma \vdash P$, *with* Γ *closed, and* $LHO\pi^- \triangleright \Gamma \vdash a\langle v\rangle, a\langle b\rangle$.

1. *If* $P \xrightarrow{a[v]} P'$ *then there are* P'', R, x *such that* $[\![P]\!]^\Gamma \xrightarrow{a[b]} P''$ *with* $P' = R\{v/x\}$ *and* $P'' = [\![R]\!]^\Gamma\{b/x\}$;
2. *the converse, i.e., if* $[\![P]\!]^\Gamma \xrightarrow{a[b]} P''$, *then there is* P', R, x *such that* $P \xrightarrow{a[v]} P'$ *with* $P' = R\{v/x\}$ *and* $P'' = [\![R]\!]^\Gamma\{b/x\}$.

If v is an abstraction or a channel, we define process v_a thus:

$$v_a \stackrel{\text{def}}{=} \begin{cases} a(x).\,P & \text{if } v = (x : T).\,P \\ a(x).\,b\langle x\rangle & \text{if } v = b \end{cases}$$

Lemma 5.6 (operational correspondence of $[\![\,.\,]\!]$ on higher-order output actions) *Suppose* $LHO\pi^- \triangleright \Gamma \vdash P$, *with* Γ *closed, and* $\mathcal{O}_\Delta(a) \equiv T \to \mathcal{B}$.

1. *If* $P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\, a\langle v\rangle} P'$, *then there is* c *fresh for* Γ *and* $\widetilde{b}$, *and there is* P'' *such that* $[\![P]\!]^\Gamma \xrightarrow{(\boldsymbol{\nu}c:T\to\mathcal{B})\, a\langle c\rangle} P''$ *with* $L\pi^- \triangleright \Gamma, c : T \to \mathcal{B} \vdash P'' \simeq [\![(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\,(P' \mid\, !v_c)]\!]^{\Gamma, c:T\to\mathcal{B}}$;
2. *the converse, i.e., if* $[\![P]\!]^\Gamma \xrightarrow{\mu} P''$ *and* μ *is an output action at* a, *then* $\mu = (\boldsymbol{\nu}c : T \to \mathcal{B})\, a\langle c\rangle$, *for some* c, *and, assuming* c *fresh for* Γ, *there are* $\widetilde{b}$, $\widetilde{T}$, v *and* P' *such that* $P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\, a\langle v\rangle} P'$ *and* $L\pi^- \triangleright \Gamma, c : T \to \mathcal{B} \vdash P'' \simeq [\![(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\,(P' \mid\, !v_c)]\!]^{\Gamma, c:T\to\mathcal{B}}$.

Lemma 5.7 *Suppose* $LHO\pi^- \triangleright \Gamma, y : T \to \mathcal{B} \vdash P$, *and* $LHO\pi^- \triangleright \Gamma \vdash v : T \to \mathcal{B}$. *Then, if* c *is fresh for* Γ,

$$L\pi^- \triangleright \Gamma \vdash [\![(\boldsymbol{\nu}c : T \to \mathcal{B})\,(P\{c/y\} \mid\, !v_c)]\!]^\Gamma \simeq [\![P\{v/y\}]\!]^\Gamma \ .$$

Proof: By induction on the order n of T (the level of arrow-nesting in T). Both in the case $n = 0$ and $n > 0$ we proceed by induction on the structure of P. The most interesting case is application when $n > 0$. We consider the subcases of $P = y\langle w \rangle$ with $w \stackrel{\text{def}}{=} (x).\, Q$ (the function is y and the argument an abstraction; we omit type annotations), and of $P = w\langle y \rangle$ (the argument is y). Below we use the laws of Lemmas 3.8 and laws (1-6) and (8) of Lemma 3.9 since, by Remark 3.10, they are also valid for ground bisimulation (laws (1-2) are only used under the restrictions mentioned in the remark). If $P = y\langle w \rangle$ we have:

$$[\![\boldsymbol{\nu} c\, (c\langle w\{^c\!/\!_y\}\rangle \mid !v_c)]\!] \quad \simeq \quad (\boldsymbol{\nu} b, c\,)(c\langle b\rangle \mid !b(x).\, [\![Q\{^c\!/\!_y\}]\!] \mid ![\![v_c]\!])\ .$$

We now proceed using the laws of the lemmas and the induction on the structure:

$$\begin{array}{rl} \simeq & (\boldsymbol{\nu} b\,)(\boldsymbol{\nu} c\, (c\langle b\rangle \mid ![\![v_c]\!]) \mid \boldsymbol{\nu} c\, (!b(x).\, [\![Q\{^c\!/\!_y\}]\!] \mid ![\![v_c]\!])) \\ \simeq & (\boldsymbol{\nu} b\,)(\boldsymbol{\nu} c\, (c\langle b\rangle \mid ![\![v_c]\!]) \mid !b(x).\, \boldsymbol{\nu} c\, ([\![Q\{^c\!/\!_y\}]\!] \mid ![\![v_c]\!])) \\ \simeq & (\boldsymbol{\nu} b, c\,)([\![v_c]\!] \mid c\langle b\rangle \mid ![\![w\{^v\!/\!_y\}]\!]_b) \\ = & [\![(y\langle w\rangle)\{^v\!/\!_y\}]\!] \\ = & [\![P\{^v\!/\!_y\}]\!]\ . \end{array}$$

For $P = w\langle y\rangle$, the crux is to prove that

$$\boldsymbol{\nu} c\, ([\![c_a]\!] \mid ![\![v_c]\!]) \simeq [\![v_a]\!]\ .$$

We have

$$\boldsymbol{\nu} c\, ([\![c_a]\!] \mid ![\![v_c]\!]) \quad \simeq \quad \boldsymbol{\nu} c\, (a(z).\, \boldsymbol{\nu} b\, (c\langle b\rangle \mid ![\![z_b]\!]) \mid ![\![v_c]\!])\ .$$

We now proceed using the laws of the lemmas and the fact that $[\![v_c]\!] = c(y).\, [\![R]\!]$, for some R:

$$\begin{array}{rl} \simeq & a(z).\, \boldsymbol{\nu} b\, (\boldsymbol{\nu} c\, (c\langle b\rangle \mid ![\![v_c]\!]) \mid ![\![z_b]\!]) \\ \simeq & a(z).\, \boldsymbol{\nu} b\, ([\![R]\!]\{^b\!/\!_y\} \mid ![\![z_b]\!]) \end{array}$$

Since z is the argument of a and a has type T, the order of the type of z is smaller than the order of T, that is n. We can therefore apply the inductive hypothesis on the types and obtain

$$\begin{array}{rl} \simeq & a(z).\, [\![R]\!]\{^z\!/\!_y\} \\ = & [\![a(y).\, R]\!] \\ = & [\![v_a]\!] \end{array}$$

■

We can now present the operational correctness of $[\![\,.\,]\!]$ on interactions.

Theorem 5.8 (operational correspondence of $[\![\,.\,]\!]$ on strong interactions) *Suppose $LHO\pi^- \triangleright \Gamma \vdash P$ and Γ is closed.*

1. *If $P \xrightarrow{\tau} P'$, then there is P'' such that $[\![P]\!]^\Gamma \xrightarrow{\tau} P''$ and $L\pi^- \triangleright \Gamma \vdash P'' \simeq [\![P']\!]^\Gamma$;*
2. *the converse, i.e., if $[\![P]\!]^\Gamma \xrightarrow{\tau} P''$, then there is P' such that $P \xrightarrow{\tau} P'$ and $L\pi^- \triangleright \Gamma \vdash P'' \simeq [\![P']\!]^\Gamma$.*

Proof: Another transition induction. In the basic case (rule COM) one needs Lemmas 5.4-5.7. ■

Lemmas 5.4-5.6 and Theorem 5.8 can be composed so to get operational correspondence on weak transitions[1]. We only report the result for weak internal transitions.

Theorem 5.9 (operational correspondence of $[\![\,.\,]\!]$ on weak interactions) *Suppose LHOπ^- $\triangleright \Gamma \vdash P$ and Γ is closed.*

1. *If $P \stackrel{\tau}{\Longrightarrow} P'$, then there is P'' such that $[\![P]\!]^\Gamma \stackrel{\tau}{\Longrightarrow} P''$ and L$\pi^- \triangleright \Gamma \vdash P'' \simeq [\![P']\!]^\Gamma$;*
2. *the converse, i.e., if $[\![P]\!]^\Gamma \stackrel{\tau}{\Longrightarrow} P''$, then there is P' such that $P \stackrel{\tau}{\Longrightarrow} P'$ and L$\pi^- \triangleright \Gamma \vdash P'' \simeq [\![P']\!]^\Gamma$.*

Corollary 5.10 (adequacy of $[\![\,.\,]\!]$) *Suppose LHOπ^- $\triangleright \Gamma \vdash P$ and Γ is closed. Then LHOπ^- $\triangleright P \Downarrow_a$ iff L$\pi^- \triangleright [\![P]\!]^\Gamma \Downarrow_a$, for all a.*

A corollary of Theorems 3.7 and 5.9 and Corollary 5.10 is the full abstraction for barbed bisimulation.

Corollary 5.11 *Suppose LHOπ^- $\triangleright \Gamma \vdash P\ ,Q$ and Γ is closed. It holds that LHOπ^- $\triangleright P \stackrel{.}{\approx} Q$ iff L$\pi^- \triangleright [\![P]\!]^\Gamma \stackrel{.}{\approx} [\![Q]\!]^\Gamma$.*

From Corollary 5.11 and the compositionality of the encoding we derive soundness for barbed congruence:

Lemma 5.12 (soundness of $[\![\,.\,]\!]$) *Suppose LHOπ^- $\triangleright \Gamma \vdash P, Q$. Then L$\pi^- \triangleright \Gamma \vdash [\![P]\!]^\Gamma \approx [\![Q]\!]^\Gamma$ implies LHOπ^- $\triangleright \Gamma \vdash P \approx Q$.*

Proof: We can extend the compilation to contexts by mapping the hole of a context onto itself. Since the encoding is compositional and respects types, for all P with LHOπ^- $\triangleright \Delta \vdash P$ and (Γ/Δ)-context C, we have $[\![C[P]]\!]^\Gamma = [\![C]\!]^\Gamma[[\![P]\!]^\Delta]$. From this, the assertion of the theorem follows by Corollary 5.11. ■

Completeness, the converse of soundness, is generally a harder result for a translation. The proof technique for soundness does not work because there may be contexts of the target language that are not translations of contexts of the source language.

Remark 5.13 Consider the subset of LHOπ^- processes in which only process values may be communicated. Then all values have type $\mathtt{unit} \rightarrow \mathcal{B}$. This calculus is translated into a subset of Lπ in which channels may only have type $(\mathtt{unit} \rightarrow \mathcal{B}) \rightarrow \mathcal{B}$ or $\mathtt{unit} \rightarrow \mathcal{B}$. The latter are "CCS names"—they carry nothing; the former are names that carry "CCS names". Aside from asynchronous, rather than synchronous, communications, this subset of Lπ goes little beyond CCS. This shows that if we only have communication of process values, then process passing gives us little more expressiveness than CCS.

[1]For composing the results we actually need an asymmetric version of $\simeq$, along the style of the expansion relation [38].

Something that cannot be done when processes are the only transmittable values is to communicate the *partial visibility* of a process. This, by contrast, is easy to model using name-passing. For instance, consider a buffer process with two channels `put` and `get` for adding and removing values. Using name-passing we can choose to transmit only the `put` or only the `get` channel; thus the recipient will have only partial access to the buffer. This is not possible using process values, where either we transmit the whole buffer or we transmit nothing. Also, when processes are the only transmittable values it is difficult to model sharing; for instance a resource that is shared by a dynamically changing set of clients. ■

5.1 Some laws for LHOπ^-

We can exploit the soundness of the encoding to prove laws for barbed congruence in LHOπ^-. For instance:

$$\begin{array}{rcl} \mathrm{LHO}\pi^- \triangleright \Gamma \vdash \; P \mid Q & \approx & Q \mid P \\ \mathrm{LHO}\pi^- \triangleright \Gamma \vdash \; ((x:T).\, v)\langle w\rangle & \approx & v\{^{w}\!/\!_{x}\} \\ \mathrm{LHO}\pi^- \triangleright \Gamma \vdash \; d & \approx & (x:T).\, d\langle x\rangle \; . \end{array}$$

The second law is a form of β-conversion; the third law a form of η-conversion on channels. As an example, we consider the proof of the third law. Let $\Delta \stackrel{\mathrm{def}}{=} \Gamma, a : (T \to \mathcal{B}) \to \mathcal{B}$. By Lemma 4.9 and Lemma 5.12, it suffices to prove that

$$\mathrm{L}\pi^- \triangleright \Delta \vdash [\![a\langle d\rangle]\!]^{\Delta} \approx [\![a\langle (x:T).\, d\langle x\rangle\rangle]\!]^{\Delta}.$$

This is true because $[\![a\langle d\rangle]\!]^{\Delta} = [\![a\langle (x:T).\, d\langle x\rangle\rangle]\!]^{\Delta}$.

5.2 Full abstraction

We derive full abstraction from Lemma 5.12 and the lemmas below.

Lemma 5.14 *Suppose Lπ^- $\triangleright \Gamma \vdash P, Q$. Then LHOπ^- $\triangleright \Gamma \vdash P \approx Q$ implies Lπ^- $\triangleright \Gamma \vdash P \approx Q$.*

Proof: LHOπ^- is an extension of Lπ^-. Every context of Lπ^- is also a context of LHOπ^-. ■

Lemma 5.15 *Lπ^- $\triangleright \Gamma \vdash p\langle q\rangle \approx [\![p\langle q\rangle]\!]^{\Gamma}$.*

Proof: By Lemmas 5.14 and 5.12, it suffices to prove that Lπ^- $\triangleright \Gamma \vdash [\![p\langle q\rangle]\!]^{\Gamma} \approx [\![[\![p\langle q\rangle]\!]^{\Gamma}]\!]^{\Gamma}$. This holds because $[\![p\langle q\rangle]\!]^{\Gamma} = [\![[\![p\langle q\rangle]\!]^{\Gamma}]\!]^{\Gamma}$. ■

Lemma 5.16 *LHOπ^- $\triangleright \Gamma \vdash P \approx [\![P]\!]^{\Gamma}$.*

Proof: By Lemma 5.12, it suffices to prove Lπ^- $\triangleright \Gamma \vdash [\![P]\!]^{\Gamma} \approx [\![[\![P]\!]^{\Gamma}]\!]^{\Gamma}$; this allows us to use the theory of Lπ^- (proving the result directly would be hard; a difficult case is that of $P = a\langle v\rangle$ and v is an abstraction). We proceed by induction on the structure of P. The most interesting case is application $P = v\langle w\rangle$ where w has a functional type. Suppose both v and w are abstractions, say $v = (x).\, Q$ and $w = (z).\, R$ (we omit types). We have

$$\begin{array}{rcl} [\![v\langle w\rangle]\!] & = & (\boldsymbol{\nu} a, b\,)(a(x).\, [\![Q]\!] \mid a\langle b\rangle \mid\, !b(z).\, [\![R]\!]) \\ [\![[\![v\langle w\rangle]\!]]\!] & = & (\boldsymbol{\nu} a, b\,)(a(x).\, [\![[\![Q]\!]]\!] \mid [\![a\langle b\rangle]\!] \mid\, !b(z).\, [\![[\![R]\!]]\!]) \; . \end{array}$$

Applying Lemma 5.15 and induction, we derive $[\![v\langle w\rangle]\!] \approx [\![[\![v\langle w\rangle]\!]]\!]$. ■

Theorem 5.17 (full abstraction of $[\![\,.\,]\!]$) *Suppose* $LHO\pi^- \triangleright \Gamma \vdash P, Q$ *. It holds that* $LHO\pi^- \triangleright \Gamma \vdash P \approx Q$ *iff* $L\pi^- \triangleright \Gamma \vdash [\![P]\!]^\Gamma \approx [\![Q]\!]^\Gamma$*.*

Proof: The implication from right to left is Lemma 5.12; for the opposite, suppose $\mathrm{LHO}\pi^- \triangleright \Gamma \vdash P \approx Q$. By Lemma 5.16, also $\mathrm{LHO}\pi^- \triangleright \Gamma \vdash [\![P]\!]^\Gamma \approx [\![Q]\!]^\Gamma$; by Proposition 5.3 $\mathrm{L}\pi^- \triangleright \Gamma \vdash [\![P]\!]^\Gamma, [\![Q]\!]^\Gamma$. Therefore, by Lemma 5.14, $\mathrm{L}\pi^- \triangleright \Gamma \vdash [\![P]\!]^\Gamma \approx [\![Q]\!]^\Gamma$. ■

6 A labeled bisimilarity for LHOπ

We exploit the compilation $[\![\,.\,]\!]$, and the associated operational correspondence and full abstraction results, to derive a labeled bisimilarity for LHOπ^-, prove that: it is a congruence relation; it implies barbed congruence. We call this bisimilarity *LHOπ ground bisimilarity,* because its clauses are designed following the definition of $[\![\,.\,]\!]$ and of ground bisimilarity for Lπ (a similar relation is called *normal bisimilarity* in [32, 34]).

Definition 6.1 (ground bisimilarity in LHOπ) *A typed LHOπ relation $\mathcal{R}$ is a LHOπ ground bisimulation if $P \; \mathcal{R}_\Delta \; Q$ implies:*

1. *if $P \xrightarrow{\tau} P'$, then there is Q' such that $Q \Longrightarrow Q'$ and $P' \; \mathcal{R}_\Delta \; Q'$;*
2. *if $\mathcal{O}_\Delta(a) \equiv \mathtt{unit}$ and $P \xrightarrow{\mu} P'$ with $\mu = a[\star]$ or $\mu = a\langle\star\rangle$, then there is Q' such that $Q \stackrel{\mu}{\Longrightarrow} Q'$ and $P' \; \mathcal{R}_\Delta \; Q'$;*
3. *if $\mathcal{O}_\Delta(a) \equiv T \to \mathcal{B}$ and $P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\, a\langle v\rangle} P'$, then there are $\widetilde{c}, \widetilde{S}, w$ and Q' such that $Q \stackrel{(\boldsymbol{\nu}\widetilde{c}:\widetilde{S})\, a\langle w\rangle}{\Longrightarrow} Q'$ and, if d is fresh for Δ, $\widetilde{b}$, and $\widetilde{c}$, then $(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\,(!v_d \mid P') \; \mathcal{R}_{\Delta, d:T\to\mathcal{B}} \; (\boldsymbol{\nu}\widetilde{c} : \widetilde{S})\,(!w_d \mid Q')$;*
4. *if $\mathcal{O}_\Delta(a) \equiv T \to \mathcal{B}$, d is fresh for Δ, and $P \xrightarrow{a[d]} P'$, then there is Q' such that $Q \stackrel{a[d]}{\Longrightarrow} Q'$ and $P' \; \mathcal{R}_{\Delta, d:T\to\mathcal{B}} \; Q'$.*

Two LHOπ (resp. LHOπ^-) processes P and Q are ground bisimilar *at Δ, written $LHO\pi \triangleright \Delta \vdash P \simeq Q$ (resp. $LHO\pi^- \triangleright \Delta \vdash P \simeq Q$), if $P \; \mathcal{R}_\Delta \; Q$, for some LHO$\pi$ ground bisimulation $\mathcal{R}$.*

The extension of ground bisimilarity to open processes is defined employing only fresh channels and unit values, in the same way as we did for Lπ. As the definition in Lπ, so ground bisimilarity in LHOπ does not have universal quantifications on contexts or on the values that may be received in inputs. This contrasts with bisimilarity for higher-order sequential calculi, like λ-calculus [1] or object calculi [14]; for instance, in Abramsky's applicative bisimilarity for the λ-calculus, two closed terms $\lambda x.\, M$ and $\lambda x.\, N$ are equivalent if, *for all closed terms L*, the terms $M\{L/x\}$ and $N\{L/x\}$ are equivalent. Again, the data independence property for LHOπ^- holds because the value received may only be used in applications and application is asynchronous. Also clause (3), for higher-order outputs, may be seen as a form of data independency, see Remark 6.2 below.

Remark 6.2 In Section 4.1 we have suggested a functional formulation of LHOπ in which the target of input and output transitions is a functional value. If we adopt this formulation, then it is natural to require that two closed functional values w_1 and w_2 of type $T \to \mathcal{B}$ are equivalent if $w_1\langle v\rangle$ and $w_2\langle v\rangle$ are equivalent, for all closed v of type T. Thus the output clause of bisimilarity becomes:

- if $\mathcal{O}_\Delta(a) \equiv T \to \mathcal{B}$ and $P \xrightarrow{\overline{a}} w_P$, then there is w_Q such that $Q \stackrel{\overline{a}}{\Longrightarrow} w_Q$ and $w_P\langle v\rangle\ \mathcal{R}_\Delta\ w_Q\langle v\rangle$ for all v with $\emptyset \vdash v\ : T$.

Here v represents the context in which the values emitted in the output is used. Similarly, the input clause becomes:

- if $\mathcal{O}_\Delta(a) \equiv T \to \mathcal{B}$ and $P \xrightarrow{a} w_P$, then there is w_Q with $Q \stackrel{a}{\Longrightarrow} w_Q$ and $w_P\langle v\rangle\ \mathcal{R}_\Delta\ w_Q\langle v\rangle$ for all v such that $\emptyset \vdash v\ : T$.

Such a bisimilarity is, essentially, *context bisimilarity*, a bisimulation for higher-order process calculi proposed in [32, 34]. We believe that—once the functional formulation of LHOπ is completed—one can prove that context bisimilarity and ground bisimilarity coincide (exploiting Corollary 6.5 below and along the lines of similar characterisation results in [32, 34]). In context bisimilarity both input and output clauses contain universal quantifications, which are removed in ground bisimilarity (the data independence property). ■

If $\mathcal{R}$ is a typed LHOπ relation, then we define:

$$
\begin{array}{rcl}
[\![\mathcal{R}]\!] & \stackrel{\text{def}}{=} & \{(\Gamma\ ;\ [\![P]\!]^\Gamma\ ;\ [\![Q]\!]^\Gamma)\ :\ \ (\Gamma\ ;\ P\ ;\ Q) \in \mathcal{R}\} \\
\simeq [\![\mathcal{R}]\!] \simeq & \stackrel{\text{def}}{=} & \{(\Gamma\ ;\ P\ ;\ Q)\ :\ \exists\ P' \text{ and } Q' \text{ with} \\
 & & \mathrm{L}\pi \triangleright \Gamma \vdash P \simeq [\![P']\!]^\Gamma, \mathrm{L}\pi \triangleright \Gamma \vdash Q \simeq [\![Q']\!]^\Gamma,\ (\Gamma\ ;\ P'\ ;\ Q') \in \mathcal{R}\}
\end{array}
$$

A typed LHOπ^- relation is a ground bisimulation iff its Lπ image is so (up to ground bisimulation).

Lemma 6.3 *Let $\mathcal{R}$ be a typed LHOπ^- relation.*

1. *if $\mathcal{R}$ is a LHOπ ground bisimulation, then $\simeq [\![\mathcal{R}]\!] \simeq$ is a Lπ ground bisimulation;*
2. *if $\simeq [\![\mathcal{R}]\!] \simeq$ is a Lπ ground bisimulation, then $\mathcal{R}$ is a LHOπ ground bisimulation.*

Lemma 6.3 is the crux for proving the correspondence between the ground bisimilarities in Lπ^- and of LHOπ^-:

Theorem 6.4 *Suppose LHOπ^- $\triangleright\ \Delta \vdash P, Q$. Then LHOπ^- $\triangleright\ \Delta \vdash P \simeq Q$ iff L$\pi \triangleright \Delta \vdash [\![P]\!]^\Delta \simeq [\![Q]\!]^\Delta$.*

Theorem 6.4 allows us to lift the known properties of Lπ ground bisimilarity onto LHOπ^-. For instance, we can prove that in LHOπ^- ground bisimilarity is preserved by all well-typed contexts and by channel renaming:

Corollary 6.5 *In LHOπ^- ground bisimilarity is a congruence relation. That is, if LHOπ^- $\triangleright \Delta \vdash P \simeq Q$, then for all (Γ/Δ) – context C of LHOπ^- it holds that LHOπ^- $\triangleright \Gamma \vdash C[P] \simeq C[Q]$.*

Corollary 6.6 *Suppose* $\Gamma(d) \equiv \Gamma(a)$. *Then* $LHO\pi^- \triangleright \Delta \vdash P \simeq Q$ *implies* $LHO\pi^- \triangleright \Delta \vdash P\{a/d\} \simeq Q\{a/d\}$.

LHOπ ground bisimilarity is a sound proof technique for barbed congruence:

Theorem 6.7 *If* $LHO\pi^- \triangleright \Delta \vdash P \simeq Q$, *then also* $LHO\pi^- \triangleright \Delta \vdash P \approx Q$.

Proof: Follows from congruence of ground bisimilarity in LHOπ^- (Corollary 6.5) and the fact that ground bisimilarity implies barbed bisimulation. ∎

We think that also the converse of Theorem 6.7 holds, but we leave it as an open problem. As $[\![\,.\,]\!]$ is fully abstract both for barbed congruence (Theorem 5.17) and for ground bisimilarity (Theorem 6.4), the converse is true iff on the target processes of $[\![\,.\,]\!]$ ground bisimilarity coincides with barbed congruence.

7 An optimisation of the compilation

In this section we introduce a simple but important optimisation of the compilation. Using the optimisation, full abstraction can be extended to recursive types and *strong* barbed congruence.

Consider the translation of an application $v\langle r\rangle$ where the argument is a name r of functional type:

$$[\![v\langle r\rangle]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu} a, b\,)([\![v]\!]_a \mid a\langle b\rangle \mid !b(x).\, [\![r\langle x\rangle]\!]) \ .$$

The translation introduces a new name b that is used as a trigger for r. Trigger b introduces a level of indirection for reaching r. The optimisation eliminates this level, by directly communicating the name r itself at a. We may call this optimisation *η-contraction on names*, because it avoids us the η-expansions on names in the clause $[\![q]\!]_a^\Gamma$ of Table 5 (this would actually be *infinite* η-expansions when the type of the name is recursive).

We call $\{\![\,.\,]\!\}$ the optimised compilation. We have introduced and studied the non-optimised compilation $[\![\,.\,]\!]$ first because it is simpler to use in proofs than the optimised $\{\![\,.\,]\!\}$. For instance, in the case of $[\![\,.\,]\!]$ we have been able to state the operational correspondence results up to ground bisimilarity (Lemma 5.4-Theorem 5.9). For $\{\![\,.\,]\!\}$, instead, we have to use barbed congruence, which is harder than ground bisimilarity to manipulate. For this reason the soundness of $[\![\,.\,]\!]$ (if not the full abstraction) can be extended or adapted to other calculi, including synchronous calculi, more smoothly than that of $\{\![\,.\,]\!\}$. Moreover using $[\![\,.\,]\!]$ it is likely that we succeed in proving an equivalence LHO$\pi \triangleright \Gamma \vdash P \approx Q$ by showing that their Lπ translations are ground bisimilar processes (as mentioned at the end of Section 6, it may actually be that on the target processes of $[\![\,.\,]\!]$ ground bisimilarity coincides with barbed congruence).

The definition of the optimised compilation on application is in Table 6; on the other operators, the definition is the same as that of the original compilation. In the table, we say that w is π-transmittable if w is a name or $w = \star$. Briefly, $\{\![\,.\,]\!\}$ is defined by replacing in the definition of application of $[\![\,.\,]\!]$, the conditions "$S \equiv \texttt{unit}$" with "w is π-transmittable", and "S is a functional type" with "w is

$$
\{\![v\langle w\rangle]\!\}^{\Gamma} \stackrel{\text{def}}{=} \begin{cases} v\langle w\rangle & \text{if } v \text{ is a name and } w \text{ is } \pi\text{-transmittable} \\ (\boldsymbol{\nu} a : S \to \mathcal{B})\,(\{\![v]\!\}^{\Gamma}_{a} \mid a\langle w\rangle) & \\ \qquad \text{if } v \text{ is not a name and } w \text{ is } \pi\text{-transmittable} & \\ (\boldsymbol{\nu} a : S \to \mathcal{B})\,(v\langle a\rangle \mid !\{\![w]\!\}^{\Gamma}_{a}) & \\ \qquad \text{if } v \text{ is a name and } w \text{ is not } \pi\text{-transmittable} & \\ (\boldsymbol{\nu} a : S \to \mathcal{B}, b : S\,)(\{\![v]\!\}^{\Gamma}_{a} \mid a\langle b\rangle \mid !\{\![w]\!\}^{\Gamma}_{b}) & \text{otherwise} \end{cases}
$$
where a, b are fresh for v, w, and $\Gamma \vdash w\ : S$.

Table 6: The definition of application for the compilation $\{\![\,.\,]\!\}$

---

not π-transmittable". As $[\![\,.\,]\!]$, so $\{\![\,.\,]\!\}$ is the identity on types. $\{\![\,.\,]\!\}$ is defined on the whole LHOπ, including recursive types. In the clauses for names and applications, the compilation looks up the types of certain values, such as the type S of w in the clause for $v\langle w\rangle$. These types are unique only up to type equality (Lemma 4.5). Since we want to treat $\{\![\,.\,]\!\}$ as a function, we assume that the compilation selects a canonical typing derivation whenever the type of a value is needed. This is a harmless decision: the processes obtained for different choices of these canonical typing derivations are the same modulo type equality.

We discuss the operational correspondence results for the optimised compilation on strong transitions. The assertions for input actions, and for outputs of the unit value are as for $[\![\,.\,]\!]$ (Lemmas 5.4 and 5.5; just replace LHOπ^- with LHOπ, and $[\![\,.\,]\!]$ with $\{\![\,.\,]\!\}$). The operational correspondence on higher-order output actions changes thus:

Lemma 7.1 (operational correspondence of $\{\![\,.\,]\!\}$ on higher-order output actions) *Suppose LHO$\pi \rhd \Gamma \vdash P$, with Γ closed, and $\mathcal{O}_{\Delta}(a) \equiv T \to \mathcal{B}$.*

1. *Suppose $P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\ a\langle v\rangle} P'$. If v is a channel, then $\{\![P]\!\}^{\Gamma} \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\ a\langle v\rangle} \{\![P']\!\}^{\Gamma}$; if v is an abstraction, then there is c fresh for Γ and $\widetilde{b}$, and there is P'' such that $\{\![P]\!\}^{\Gamma} \xrightarrow{(\boldsymbol{\nu} c:S)\ a\langle c\rangle} P''$ with $S \equiv T \to \mathcal{B}$, and $L\pi \rhd \Gamma, c : T \to \mathcal{B} \vdash P'' \sim \{\![(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\ (P' \mid !v_c)]\!\}^{\Gamma, c:T\to\mathcal{B}}$;*

2. *the converse, i.e., if $[\![P]\!]^{\Gamma} \xrightarrow{(\boldsymbol{\nu}\widetilde{c}:\widetilde{S})\ a\langle c\rangle} P''$ then either there is P' such that $P \xrightarrow{(\boldsymbol{\nu}\widetilde{c}:\widetilde{S})\ a\langle c\rangle} P'$ and $P'' = \{\![P']\!\}^{\Gamma}$; or $\widetilde{c} = c$ and, assuming c fresh for Γ, there are $\widetilde{b}$, $\widetilde{T}$, v and P' such that $P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}:\widetilde{T})\ a\langle v\rangle} P'$, $\widetilde{S} \equiv T \to \mathcal{B}$, and $L\pi \rhd \Gamma, c : T \to \mathcal{B} \vdash P'' \sim \{\![(\boldsymbol{\nu}\widetilde{b} : \widetilde{T})\ (P' \mid !v_c)]\!\}^{\Gamma, c:T\to\mathcal{B}}$.*

In the case of internal transitions, in contrast with the original compilation $[\![\,.\,]\!]$, we can now prove a 1-to-1 correspondence up to *strong* barbed congruence. This is possible because we can use the strong equivalence in the new version of Lemma 5.7 (and, moreover, we only need the lemma in the case v is an abstraction):

Lemma 7.2 *Suppose* $LHO\pi \triangleright \Gamma, y : T \to \mathcal{B} \vdash P$ *, and* $LHO\pi \triangleright \Gamma \vdash v : T \to \mathcal{B}$ *, where* v *is an abstraction. Then, if* c *is fresh for* Γ*,*

$$L\pi \triangleright \Gamma \vdash \{\![(\boldsymbol{\nu} c : T \to \mathcal{B})\,(P\{^{c}\!/\!_{y}\} \mid !v_c)]\!\}^{\Gamma} \sim \{\![P\{^{v}\!/\!_{y}\}]\!\}^{\Gamma} .$$

Proof: The proof is similar to that of Lemma 5.7, and goes by induction on the structure of P. One of the main differences is in the case of application if $P = y\langle y \rangle$ and v is an abstraction. Using law (7) of Lemma 3.9, we have

$$\boldsymbol{\nu} c\,(c\langle c \rangle \mid !\{\![v_c]\!\}) \quad \sim \quad (\boldsymbol{\nu} c, c'\,)(c\langle c' \rangle \mid !\{\![v_c]\!\} \mid !\{\![v_{c'}]\!\}) .$$

Since name c is used only once, and in output subject position, we can apply law (8) of Lemma 3.9 to remove the replication in front of $\{\![v_c]\!\}$. Rearranging terms, we get

$$\begin{array}{ll} \sim & (\boldsymbol{\nu} c, c'\,)(\{\![v_c]\!\} \mid c\langle c' \rangle \mid !\{\![v_{c'}]\!\}) \\ = & \{\![v\langle v \rangle]\!\} . \end{array}$$

■

Theorem 7.3 (operational correspondence for $\{\![\,.\,]\!\}$ on strong interactions) *Suppose* $LHO\pi \triangleright \Gamma \vdash P$ *and* Γ *is closed.*

1. *If* $P \xrightarrow{\tau} P'$*, then there is* P'' *such that* $\{\![P]\!\}^{\Gamma} \xrightarrow{\tau} P''$ *and* $L\pi \triangleright \Gamma \vdash P'' \sim \{\![P']\!\}^{\Gamma}$*;*
2. *the converse, i.e., if* $\{\![P]\!\}^{\Gamma} \xrightarrow{\tau} P''$*, then there is* P' *such that* $P \xrightarrow{\tau} P'$ *and* $L\pi \triangleright \Gamma \vdash P'' \sim \{\![P']\!\}^{\Gamma}$*.*

Lemma 7.4 (soundness of $\{\![\,.\,]\!\}$ for strong barbed congruence) *Suppose* $LHO\pi \triangleright \Gamma \vdash P, Q$ *. Then* $L\pi \triangleright \Gamma \vdash \{\![P]\!\}^{\Gamma} \sim \{\![Q]\!\}^{\Gamma}$ *implies* $LHO\pi \triangleright \Gamma \vdash P \sim Q$*.*

Lemma 7.5 $LHO\pi \triangleright \Gamma \vdash P \sim \{\![P]\!\}^{\Gamma}$*.*

Proof: Follows from soundness, since the encoding is idempotent. ■

Theorem 7.6 (full abstraction of $\{\![\,.\,]\!\}$ for strong barbed congruence) *Suppose* $LHO\pi \triangleright \Gamma \vdash P, Q$ *. It holds that* $LHO\pi \triangleright \Gamma \vdash P \sim Q$ *iff* $L\pi \triangleright \Gamma \vdash \{\![P]\!\}^{\Gamma} \sim \{\![Q]\!\}^{\Gamma}$*.*

Proof: The implication from right to left is Lemma 7.4; for the opposite, suppose LHO$\pi \triangleright \Gamma \vdash P \sim Q$. Then by Lemma 7.5, also LHO$\pi \triangleright \Gamma \vdash \{\![P]\!\}^{\Gamma} \sim \{\![Q]\!\}^{\Gamma}$. By the analogous of Lemma 5.14 for strong barbed congruence, L$\pi \triangleright \Gamma \vdash \{\![P]\!\}^{\Gamma} \sim \{\![Q]\!\}^{\Gamma}$. ■

Similar reasoning proves that $\{\![\,.\,]\!\}$ is fully abstract for weak barbed congruence.

Theorem 7.7 (full abstraction of $\{\![\,.\,]\!\}$ for weak barbed congruence) *Suppose* $LHO\pi \triangleright \Gamma \vdash P, Q$ *. It holds that* $LHO\pi \triangleright \Gamma \vdash P \approx Q$ *iff* $L\pi \triangleright \Gamma \vdash \{\![P]\!\}^{\Gamma} \approx \{\![Q]\!\}^{\Gamma}$*.*

If we abstract from internal work and we omit recursive types, then the original and the optimised compilation are equivalent:

Lemma 7.8 *Suppose LHOπ^- $\triangleright \Gamma \vdash P$. Then Lπ^- $\triangleright \Gamma \vdash [\![P]\!]^\Gamma \approx \{\![P]\!\}^\Gamma$.*

Proof: From Lemmas 5.16 and 7.5. ■

Using the encoding $\{\![\,.\,]\!\}$, and exploiting a form of labelled bisimilarity for Lπ more refined than ground bisimilarity [23, Section 5] (it is actually ground bisimilarity on a modified transition system), the properties of ground bisimilarity for LHOπ^- in Section 6 can be extended to LHOπ.

8 Conclusions

In this paper, we have first presented the basic theory of *Local* π (Lπ), a typed subcalculus of the π-calculus. We have discussed the syntactic differences between Lπ and π-calculus, and their impact on theory, expressiveness, and applications. We have then extended Lπ with communications of higher-order values, namely abstractions, obtaining the *Local Higher-Order* π*-calculus* (LHOπ). We have shown that the extension does not add expressive power. For this, we have proved the full abstraction of two compilations of LHOπ into Lπ. The first is defined on the subcalculus LHOπ^- without recursive types; the second is suggested by an optimisation of the first. We have used the compilations to prove the validity of some algebraic laws in LHOπ and the congruence of a labeled bisimilarity for LHOπ^-. The advantages of the optimised compilation are: the target processes are more efficient (they perform fewer τ steps than those obtained from the original compilation); full abstraction holds both for weak and for strong barbed congruence; recursive types are allowed. The optimised compilation is, however, less robust: its full abstraction may break in calculi without the syntactic constraints of Lπ. For instance, without the constraints of asynchrony, output capability, and no matching, we lose law (3) of Section 3.3, hence the processes $\{\![a\langle d\rangle]\!\}^\Gamma$ and $\{\![a\langle(x:S).\,d\langle x\rangle\rangle]\!\}^\Gamma$ are not (strong or weak) barbed congruent, for $\Gamma \stackrel{\text{def}}{=} a:(S\to\mathcal{B})\to\mathcal{B}, b:S\to\mathcal{B}$. However the source target processes remain barbed congruent at Γ. (The equivalence between $\{\![a\langle d\rangle]\!\}^\Gamma$ and $\{\![a\langle(x:S).\,d\langle x\rangle\rangle]\!\}^\Gamma$ in synchronous calculi can be regained using receptive types [36].)

In this paper we have worked with *synchronous*, rather than *asynchronous*, behavioural equivalences. For barbed congruence, the only difference in the asynchronous version is that input actions are not observable (that is, $P\downarrow_a$ only detects if P can perform an output at a); but for labeled bisimilarities, such as those discussed in Section 3.2, the asynchronous versions are more complex [3]. The choice of synchronous behavioural equivalences is not important for the results in the paper; they can be adapted to asynchronous behavioural equivalences with the expected modifications.

The compilation in Table 5 is that in [32], specialised to LHOπ^-. The completeness proof in this paper is however quite different from, and much simpler than, that in [32]. The calculi in [32] have none of the syntactic constraints of Lπ and LHOπ but, as a consequence, the full abstraction proof relies on certain infinitary features of the language such as infinite summation. These features are needed to prove characterisations of barbed congruence in π-calculus and Higher-Order π-calculus in terms of appropriate labeled bisimilarities. The compilation itself is derived in two steps, the first of which is a mapping of the

Higher-Order π-calculus into itself that "normalises" processes, that is, it transforms processes so that every value emitted in an output has a special syntactic form. Completeness is the hard part of the proof in [32]; the soundness proof similar to that of Lemma 5.12. In this paper, Lemmas 5.16 and 7.5, which are derived from the soundness of the compilations, allow us a dramatic shortcut of the completeness proofs. The compilation in [32], as its analogous in Table 5, cannot handle recursive types and is not correct for strong barbed congruence.

The compilation in [32] was in turn inspired by Thomsen's translation of Plain CHOCS into π-calculus [39, 40] (in Plain CHOCS only processes—no channels or abstractions—can be communicated). Thomsen gives an operational correspondence result on a modified transition system; he does not prove full abstraction.

Key lemmas in our full abstraction proofs are Lemmas 5.16 and 7.5, which state that a process and its translation are behaviourally equivalent. A process and its translation can be compared since they are typable in the same type environment. This is possible because, due to the syntactic constraints in Lπ (and hence also in LHOπ), the types for channels and abstractions can be the same. Even if the types of channels and abstractions were different, a similar completeness proof could be given as follows. Define a reverse compilation $\{\!\!\{\cdot\}\!\!\}$ from Lπ to LHOπ that η-expands an output $p\langle q\rangle$, where q is a channel or a channel variable, to $p\langle (x).\, q\langle x\rangle\rangle$, and that is an homomorphism everywhere else. Then prove, in place of Lemmas 5.16 and 7.5 (and, as in the proofs of these lemmas, exploiting soundness), that the composition of the compilations from LHOπ to Lπ with $\{\!\!\{\cdot\}\!\!\}$ is the identity up to barbed congruence. Again, the constraints on asynchrony, output capability, and no matching of Lπ are necessary for the definition of $\{\!\!\{\cdot\}\!\!\}$.

In both compilations in Section 5 and Section 7, the translation of application $v\langle w\rangle$ uses case analysis on the syntax or the types of v and w. A more compact, but less efficient definition can be given thus:

$$\langle\!\langle\, v\langle w\rangle \,\rangle\!\rangle^{\Gamma} \stackrel{\mathrm{def}}{=} \begin{cases} (\boldsymbol{\nu} a : \mathtt{unit} \to \mathcal{B})\,(\langle\!\langle\, v \,\rangle\!\rangle^{\Gamma}_{a} \mid a\langle w\rangle) & \text{if } \Gamma \vdash w : \mathtt{unit} \\ (\boldsymbol{\nu} a : S \to \mathcal{B}, b : S\,)(\langle\!\langle\, v \,\rangle\!\rangle^{\Gamma}_{a} \mid a\langle b\rangle \mid !\langle\!\langle\, w \,\rangle\!\rangle^{\Gamma}_{b}) & \\ & \text{otherwise, for } \Gamma \vdash w : S \end{cases}$$
$$\text{assuming } a, b \text{ fresh for } v, w$$

The other clauses for $\langle\!\langle \cdot \rangle\!\rangle$ are the same as for $[\![\,.\,]\!]$. Encoding $\langle\!\langle \cdot \rangle\!\rangle$ is less efficient than $[\![\,.\,]\!]$ (and hence $\{\![\,.\,]\!\}$) because if v is a name then $\langle\!\langle\, v\langle w\rangle \,\rangle\!\rangle^{\Gamma}$ has an initial τ-step that $[\![\,.\,]\!]$ does not have. However, abstracting from τ-steps, the two encodings are equivalent, indeed ground bisimilar: $\mathrm{L}\pi^{-} \rhd \Gamma \vdash [\![P]\!]^{\Gamma} \simeq \langle\!\langle\, Q \,\rangle\!\rangle^{\Gamma}$.

The results in the paper can be extended to calculi with richer type systems, for instance with products, variants, subtyping. It would be interesting to study the applicability of the results to other higher-order concurrent languages, for instance a π-calculus-based language such as Pict [30]. The goal would be to use the compilation to derive proof techniques for the source languages; as discussed in the Introduction, defining proof techniques directly on higher-order process languages is usually very hard.

Perhaps the most negative consequence of the syntactic constraints of Lπ is that axiomatisations of the π-calculus [29] cannot be reused. Finding a sound and complete axiomatisation of barbed congruence or other behavioural equivalences (possibly weak equivalences) on finite (i.e., replication-free) Lπ processes will probably require a quite different approach.

Acknowledgements

I thank Massimo Merro for several useful discussions on Lπ, and David Walker for comments on an early draft. The paper has changed and improved a lot as a consequence of the suggestions and remarks of the three anonymous referees; they also spotted technical weaknesses of a previous draft. I am very grateful to them for their insightful and constructive criticism.

This research has been supported by the project CONFER-2 (WG-21836).

References

[1] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[2] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[3] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[4] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118), and as DEC Systems Research Center Research Report number 62, August 1990.

[5] M. Boreale. On the expressiveness of internal mobility in name-passing calculi. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[6] G. Boudol. Towards a lambda calculus for concurrent and communicating systems. In *TAPSOFT '89*, volume 351 of *Lecture Notes in Computer Science*, pages 149–161, 1989.

[7] G. Boudol. Asynchrony and the π-calculus. Technical Report RR-1702, INRIA-Sophia Antipolis, 1992.

[8] G. Boudol. The pi-calculus in direct style. In *Proc. 24th POPL*. ACM Press, 1997.

[9] J. Despeyroux. Higher-order specification of the pi-calculus. In preparation, 1999.

[10] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. Tr 95:05, School of Cognitive and Computing Sciences, University of Sussex, 1995.

[11] M.J. Fischer, N.A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[12] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join calculus. In *Proc. 23th POPL*. ACM Press, 1996.

[13] A. Giacalone, P. Mishra, and S. Prasad. FACILE, a symmetric integration of concurrent and functional programming. In J. Diaz and F. Orejas, editors, *TAPSOFT '89*, volume 352 of *Lecture Notes in Computer Science*, pages 189–209. Springer Verlag, 1989.

[14] A.D. Gordon and G.D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proc. 23th POPL*. ACM Press, 1996.

[15] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial intelligence*, 8(3):323–364, 1977.

[16] K. Honda. Two bisimilarities for the ν-calculus. Technical Report 92-002, Keio University, 1992.

[17] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communications. In M. Tokoro, O. Nierstrasz, P. Wegner, and A. Yonezawa, editors, *ECOOP '91 Workshop on Object Based Concurrent Programming,* Geneva, Switzerland, 1991, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer Verlag, 1991.

[18] K. Honda and M. Tokoro. A Small Calculus for Concurrent Objects. In *OOPS Messanger*, Association for Computing Machinery. 2(2):50-54, 1991.

[19] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[20] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[21] H. Hüttel, J. Kleist, M. Merro, and U. Nestmann. Migration = cloning ; aliasing. To be presented at Sixth Workshop on Foundations of Object-Oriented Languages (FOOL 6), 1999.

[22] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107:272–302, 1993.

[23] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *25th ICALP*, volume 1443 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[24] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[25] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[26] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[27] U. Nestmann and B. Pierce. Decoding choice encodings. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[28] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Proc. 24th POPL*. ACM Press, 1997.

[29] J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120(2):174–197, 1995.

[30] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.

[31] J. Reppy. CML: A higher-order concurrent langauge. In *Programming Language Design and Implementation*. SIGPLAN, ACM, 1991.

[32] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[33] D. Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA-Sophia Antipolis, 1995. To appear in "Festschrift volume in honor of Robin Milner's 60th birthday", MIT Press.

[34] D. Sangiorgi. Bisimulation for Higher-Order Process Calculi. *Information and Computation*, 131(2):141–178, 1996.

[35] D. Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

[36] D. Sangiorgi. The name discipline of receptiveness. In *24th ICALP*, volume 1256 of *Lecture Notes in Computer Science*. Springer Verlag, 1997. To appear in TCS.

[37] D. Sangiorgi. Interpreting functions as pi-calculus processes: a tutorial. Revised version of TR RR-3470, INRIA-Sophia Antipolis. Available as ftp://zenon.inria.fr/meije/theorie-par/davides/functionPItutorial.ps.gz, 1999.

[38] D. Sangiorgi and R. Milner. The problem of "Weak Bisimulation up to". In W.R. Cleveland, editor, *Proc. CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer Verlag, 1992.

[39] B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Department of Computing, Imperial College, 1990.

[40] B. Thomsen. Plain CHOCS, a second generation calculus for higher-order processes. *Acta Informatica*, 30:1–59, 1993.

[41] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.