UNIVERSITY OF SUSSEX

COMPUTER SCIENCE

Subtyping and Locality in Distributed Higher Order Processes

Nobuko Yoshida
Matthew Hennessy

Report 1/99 June 1999

Computer Science
School of Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH

ISSN 1350–3170

Subtyping and Locality in Distributed Higher Order Processes

NOBUKO YOSHIDA and MATTHEW HENNESSY

ABSTRACT. This paper studies one important aspect of distributed systems, *locality*, using a calculus of distributed higher-order processes in which not only basic values or channels, but also parameterised processes are transferred across distinct locations. An integration of the subtyping of λ-calculus and IO-subtyping of the π-calculus offers a tractable tool to control the locality of channel names in the presence of distributed higher order processes. Using a local restriction on channel capabilities together with a subtyping relation, locality is preserved during reductions even if we allow new receptors to be dynamically created by instantiation of arbitrary higher-order values and processes.

We also show that our method is applicable to more general constraints, based on local and global channel capabilities.

1 Introduction

There have been a number of attempts at adapting traditional process calculi, such as CCS and CSP, so as to provide support for the modelling of certain aspects of distributed systems, such as *distribution* of resources and *locality*, [3, 11, 22, 28, 34]. Most of these are based on first-order extensions of the π-calculus [23]; first-order in the sense that the data exchanged between processes are from simple datatypes, such as basic values or channel names. There are various proposals for implementing the transmission of higher-order data using these first-order languages, mostly based on [31]. However these translations, as we will explain in Section 6, do not preserve the distribution and locality of the source language. Consequently we believe that higher-order extensions of the π-calculus should be developed in their own right, as formal modelling languages for distributed systems.

In this paper we design such a language and examine one important aspect of distributed systems, namely *locality*. The language is a simple conservative extension of the call-by-value λ-calculus [29] and the π-calculus [23], together with primitives for distribution and spawning of new code at remote sites. The combination of dynamic channel creation inherited from π-calculus and transmission of higher-order programs inherited from λ-calculus offers us direct descriptions of various distributed computational structures. As such, it has much in common

Research supported by EPSRC Grant K60701 and CONFER II.
E-mail: {`nobuko,matthewh`}`@cogs.susx.ac.uk`.

with the core version of Facile [2, 9, 21] and CML [8] and can be regarded as an extension of Blue-calculus [5] to a higher-order term passing language.

A desirable feature of some distributed systems is that every channel name is associated with a *unique* receptor, which is called *receptiveness* in [32]; another property called *locality* where new receptors are not created by received channels, has also been studied in [3, 4, 22, 38] for an asynchronous version of the π-calculus. The combination of these constraints provides a model of a realistic distributed environment, which regards a receptor as an object or a thread existing in a unique name space. A generalisation is also proposed in Distributed Join-calculus where not only single receptor but also several receptors with the same input channel are allowed to exist in the same location [11]; in this paper we call this more general condition *locality of channels*. In distributed object-oriented systems, objects with a given ID reside in a specific location even if multiple objects with the same ID are permitted to exist for efficiency reasons, as found for example in CONCURRENT AGGREGATES [7]. This locality constraint should be obeyed even in the presence of higher-order parameterised object passing, which is recently often found in practice [12].

In this paper we show that, in a distributed higher-order process language, locality of channels can be enforced by a typing system with subtyping. The essential idea is to control the *input capability* of channels, guaranteeing at any one time that this capability resides at exactly one location. As discussed in Section 3, ensuring locality in higher order processes is much more difficult than in systems which only allows name passing. However, using our typing system we only have to *static* type-check each local configuration to guarantee the required global invariance, namely *locality of channels*.

The main technical novelty of our work is an extension of the input/output type system of [27, 15] to a higher-order setting. We use the more expressive set of input/output types from [15], although the formalisation is somewhat different in order to have a natural integration with the arrow types of the λ-calculus. The order theoretic property of the subtyping relation, *finite-bounded completeness*, satisfied in our formalism plays a pivotal role for a technical development throughout this paper. The framework will be generally applicable for other purposes where similar global constraints should be guaranteed using static local type checking.

The paper is organised as follows. In the following section we study the undistributed version of our language, $\pi\lambda$, a call-by-value λ-calculus with communication primitives based on channels. Section 3 introduces a distributed version of $\pi\lambda$, which we call $D\pi\lambda$, by adding a process spawning operator and a primitive notion of distribution. We then explain, using examples, the difficulty of enforcing locality in $D\pi\lambda$. Section 4 gives a typing system based on the input/output typing in Section 2, which ensures locality of all channels in $D\pi\lambda$ by local static type-checking. In Section 5, we discuss applications of our work; extendibility of our typing system to more general global/local channel constraints studied by [34] in a higher-order setting, the proof of a multiple higher-order replication theorem extended from [27, 32], and the type-checking. Section 6 concludes with discussion and related work.

$$
\begin{array}{llcl}
\text{Term:} & P,Q,\ldots \in \mathit{Term} & ::= & V \mid PQ \mid \\
 & & & u?(\tilde{x}:\tilde{\tau}).P \mid u!\langle\tilde{V}\rangle P \mid (\nu a:\sigma)P \mid *P \mid \\
 & & & P|Q \mid \mathbf{0} \\
\text{Value:} & V,W,\ldots \in \mathit{Val} & ::= & u \mid \lambda(x:\tau).P \\
\text{Identifier:} & u,v,\ldots \in \mathit{Id} & ::= & l \mid a \mid x \\
\text{Literal:} & l,l',\ldots \in \mathit{Lit} & ::= & \mathtt{tt} \mid \mathtt{ff} \mid () \mid 0 \mid 1 \mid \ldots \mid 1.1 \mid \ldots
\end{array}
$$

FIGURE 1. Syntax of $\pi\lambda$

2 A Higher-order π-calculus with IO-subtyping

In this section, we introduce a higher order concurrent calculus with subtyping, essentially the call-by-value λ-calculus [29] augmented with the π-calculus primitives [23]. We illustrate the usage of this typing system by a few simple examples.

SYNTAX The syntax of $\pi\lambda$ is given in Figure 1. It uses an infinite set of *names* or *channels* **N**, ranged over by $a, b, \ldots$, and an infinite set of *variables* **V**, $x, y, \ldots$. We often use $X, Y, \ldots$ for variables over higher order terms explicitly. It also uses a collection of types, the discussion of which we defer until later.

The syntax is a mixture of a call-by-value λ-calculus and the π-calculus. From the former there are values, consisting of basic values and abstractions, together with application; the sequencing operator $\mathtt{let}\, x : \tau = P \;\mathtt{in}\; Q$ can be viewed as an alternative notation for $(\lambda(x:\tau).Q)\,P$.

From the latter we have input and output on communication channels, dynamic channel creation, iteration and the empty process. All bound variables and names have associated with them a type, but for the moment these are ignored. We use the standard notational conventions associated with the π-calculus, for example ignoring trailing occurrences of the empty process $\mathbf{0}$ and omitting type annotations unless they are relevant. We also use $\mathsf{fn}(P)/\mathsf{bn}(P)$ and $\mathsf{fv}(P)/\mathsf{bv}(P)$ to denote the sets of *free/bound names* and *free/bound variables*, to respectively, defined in the standard manner. We also assume all bound names are distinct and disjoint from free names, and typically write $\lambda().P$ for a *thunk* of P, $\lambda(x:\mathtt{unit}).P$

Reduction Rules:

$$(\beta) \qquad (\lambda(x:\tau).P)V \longmapsto P\{V/x\}$$

$$(\text{com}) \qquad u?(\tilde{x}:\tilde{\tau}).P \mid u!\langle\tilde{V}\rangle Q \longmapsto P\{\tilde{V}/\tilde{x}\} \mid Q$$

$$(\text{app}_l)\ \frac{P \longmapsto P'}{P\,V \longmapsto P'\,V} \qquad (\text{app}_r)\ \frac{Q \longmapsto Q'}{P\,Q \longmapsto P\,Q'}$$

$$(\text{par})\ \frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q} \qquad (\text{res})\ \frac{P \longmapsto P'}{(\nu a:\sigma)P \longmapsto (\nu a:\sigma)P'} \qquad (\text{str})\ \frac{P \equiv P' \longmapsto Q' \equiv Q}{P \longmapsto Q}$$

Structure Equivalence:

- $P \equiv Q$ if $P \equiv_\alpha Q$.
- $P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad P \mid \mathbf{0} \equiv P \quad *P \equiv P \mid *P$
- $(\nu a:\sigma)\mathbf{0} \equiv \mathbf{0} \quad (\nu a:\sigma)(\nu b:\sigma')P \equiv (\nu b:\sigma')(\nu a:\sigma)P.$

  $(\nu a:\sigma)P \mid Q \equiv (\nu a:\sigma)(P \mid Q)$ if $a \notin \mathsf{fn}(Q)$

FIGURE 2. Reduction for $\pi\lambda$

assuming $x \notin \mathsf{fv}(P)$.

REDUCTION The reduction semantics of $\pi\lambda$ is given in Figure 2 and is relatively straightforward. The main reduction rules are value β-reduction, (β), for the functional part of the language and communication, (com), for processes. The final contextual rule, (str), uses a structural congruence borrowed from standard presentations of the π-calculus (Figure 2). We use $\longmapsto\!\!\!\to$ to denote multi-step reductions.

EXAMPLES In this subsection, we show a couple of simple programs using $\pi\lambda$.

EXAMPLE 2.1. (Sq-server, 1) Suppose that in the language we have a literal `sq` for squaring natural numbers; this is a simple example of a data processing operation which may, in general, be quite complicated. For a given name a let $\mathbf{sq}(a)$ represent the expression $*a?(y,z).\,z!\langle \mathtt{sq}(y)\rangle$, which we write as

$$\mathbf{sq}(a) \Longleftarrow *a?(y,z).\,z!\langle \mathtt{sq}(y)\rangle$$

This receives a value on y to be processed together with a return channel z to which the processed data is to be sent. It then processes the data (in this case simply squaring it) and returns the processed data along the return channel. □

Type:

Term Type:	ρ	$::=$	$\mathtt{proc} \mid \tau$
Value Type:	τ	$::=$	$\mathtt{unit} \mid \mathtt{bool} \mid \mathtt{nat} \mid \mathtt{real} \mid \tau \to \rho \mid \sigma$
Channel Type:	σ	$::=$	$\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$ with $S_{\mathtt{I}} \geq S_{\mathtt{O}},\ S_{\mathtt{I}} \neq \bot$ and $S_{\mathtt{O}} \neq \top$.
Sort Type:	S	$::=$	$\bot \mid \top \mid (\tilde{\tau})$

Ordering:

(base)	$\mathtt{proc} \leq \mathtt{proc},\ \mathtt{nat} \leq \mathtt{nat},\ S \leq S$, etc.
$(\bot,\top)$	$\bot \leq S \quad S \leq \top$
(vec)	$\forall i.\ \tau_i \leq \tau'_i \ \Rightarrow\ (\tilde{\tau}) \leq (\tilde{\tau}')$
$(\to)$	$\tau \geq \tau',\ \rho \leq \rho' \ \Rightarrow\ \tau \to \rho \leq \tau' \to \rho'$
(chan)	$\sigma_i = \langle S_{i\mathtt{I}}, S_{i\mathtt{O}}\rangle,\ S_{1\mathtt{I}} \leq S_{2\mathtt{I}},\ S_{1\mathtt{O}} \geq S_{2\mathtt{O}} \ \Rightarrow\ \sigma_1 \leq \sigma_2.$

FIGURE 3. Types for $\pi\lambda$

Many examples of this continuation-passing style programming can be found in [28, 11]. However in our language, we can pass not only basic values and channels but also *program abstraction*, which would be instantiated by both channels and higher-order terms.

EXAMPLE 2.2. (Sq-server, 2) A sq-server is a process which on requests sends to the client the code for squaring values, which the client can initialise locally. In $\pi\lambda$ this can be defined by

$$\mathbf{sqServ} \Longleftarrow *\mathrm{req}?(r).\ r!\langle\lambda(x).\ \mathbf{sq}(x)\rangle$$

Here the process receives a request on the channel req, in the form of a return channel r, to which the abstraction $\lambda(x).\ \mathbf{sq}(x)$ is sent. A client can now download this code and initialise it by applying it to a local channel which will act as the request channel for data processing:

$$\mathbf{Client} \Longleftarrow (\nu r)\ \mathrm{req}!\langle r\rangle.\ r?(X).\ (\nu a)(\ X\,a\ \mid a!\langle 1, c_1\rangle \mid a!\langle 2, c_2\rangle \mid a!\langle 3, c_3\rangle \mid \cdots)$$

□

IO-TYPES We use as types for $\pi\lambda$ a simplification of the input/output capabilities of [15] (in turn a *strict* generalisation of [27]). They are defined in Figure 3, where we assume a given set of base types, such as `nat`, `real` and `bool`, and a type for processes, `proc`. Value types, types of objects which may be transmitted between processes or to which functions may be applied, may then be constructed from these types using the exponential type constructor $\to$, as in the λ-calculus. However here in addition we may also use channel types, ranged over by σ. These take the form $\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$, a pair consisting of an *input sort* $S_{\mathtt{I}}$ and an *output sort* $S_{\mathtt{O}}$; these input/output sorts are in turn either a vector of value types or $\top$, denoting the highest capability, or $\bot$, denoting the lowest. The representation of IO-types as a tuple [16, 15] makes the definition of the subtyping relationship, also given in Figure 3, more natural when we integrate with arrow types of the λ-calculus; the ordering of input types is covariant, whereas that of output types is contravariant. The condition on channel types, $S_{\mathtt{I}} \geq S_{\mathtt{O}}$ is necessary to ensure that a receiver can take from a channel at most the capabilities sent to it. Then, as already discussed in [15], IO-types in [27] are represented as a special case of our IO-types; to denote them, we introduce the following abbreviations.

(input only)	$(\tilde{\tau})^{\mathtt{I}}$	for $\langle(\tilde{\tau}), \bot\rangle$
(output only)	$(\tilde{\tau})^{\mathtt{O}}$	for $\langle\top, (\tilde{\tau})\rangle$
(input/output)	$(\tilde{\tau})^{\mathtt{IO}}$	for $\langle(\tilde{\tau}), (\tilde{\tau})\rangle$

Note that $(\tilde{\tau})^{\mathtt{IO}} \leq (\tilde{\tau})^{\mathtt{I}} \leq \langle\top, \bot\rangle$ and $(\tilde{\tau})^{\mathtt{IO}} \leq (\tilde{\tau})^{\mathtt{O}} \leq \langle\top, \bot\rangle$. Note also $\langle\top, \bot\rangle \neq \top$ because the former is a type for a channel which is only used as a value (i.e. empty capability), while the latter is the top of sort types. Then the key order-theoretic property of the set of types follows (The reader is referred to [15] for the definition of a finite bounded complete partial order).

PROPOSITION 2.3. (Ordering) *The subtyping relation over types defined in Figure 3 is a finite bounded complete partial order,* FBC.

Proof The proof is similar to that of Proposition 6.2 in [15] and is omitted. □

The required partial meet operator $\sqcap$ and partial join operator $\sqcup$ can be defined directly, based on the following clauses.[1]

(base)	$\rho \sqcap \rho \stackrel{\mathrm{def}}{=} \rho \sqcup \rho \stackrel{\mathrm{def}}{=} \rho$	
$(\top, \bot)$	$\bot \sqcap S \stackrel{\mathrm{def}}{=} \bot, \top \sqcup S \stackrel{\mathrm{def}}{=} \top$ and	
	$\bot \sqcup S \stackrel{\mathrm{def}}{=} \top \sqcap S \stackrel{\mathrm{def}}{=} S.$	
(abs)	$(\tau_1 \to \rho_1) \sqcap (\tau_2 \to \rho_2) \stackrel{\mathrm{def}}{=} \tau_1 \sqcup \tau_2 \to \rho_1 \sqcap \rho_2$	and
	$(\tau_1 \to \rho_1) \sqcup (\tau_2 \to \rho_2) \stackrel{\mathrm{def}}{=} \tau_1 \sqcap \tau_2 \to \rho_1 \sqcup \rho_2$	
(vec)	$(\tilde{\tau}) \sqcup (\tilde{\tau}') \stackrel{\mathrm{def}}{=} (\tilde{\tau}'')$ with $\tau''_i = \tau_i \sqcup \tau'_i$ and	
	$(\tilde{\tau}) \sqcap (\tilde{\tau}') \stackrel{\mathrm{def}}{=} (\tilde{\tau}'')$ with $\tau''_i = \tau_i \sqcap \tau'_i$	
(chan)	$\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle \sqcup \langle S'_{\mathtt{I}}, S'_{\mathtt{O}}\rangle \stackrel{\mathrm{def}}{=} \langle S_{\mathtt{I}} \sqcup S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcap S'_{\mathtt{O}}\rangle$	and
	$\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle \sqcap \langle S'_{\mathtt{I}}, S'_{\mathtt{O}}\rangle \stackrel{\mathrm{def}}{=} \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}}\rangle$	if $S_{\mathtt{I}} \geq S'_{\mathtt{O}}$ and $S'_{\mathtt{I}} \geq S_{\mathtt{O}}$; else undefined.

[1]Note that we do not include $\top$ and $\bot$ for base and arrow types for simplicity. The side coditions $S_{\mathtt{I}} \neq \bot$ and $S_{\mathtt{O}} \neq \top$ in the channel types ensure that a term which misuses arity constraints, and as $a!\langle V\rangle \mid a!\langle V_1 V_2\rangle$ can not be typed.

For sort types (but not value, term or channel types) we can ensure that both $\sqcap$ and $\sqcup$ are total; in all cases of $S \sqcap S'$ (respectively $S \sqcup S'$) not covered by the above clasues (for example if they are structually dissimilar or do not satisfy the IO constraint), then we set $S \sqcap S' = \bot$ (respectively $S \sqcup S' = \top$).

THE IO TYPING SYSTEM *Type environments*, ranged over by $\Gamma, \Delta, \ldots$, are functions from a finite subset of $\mathbf{N} \cup \mathbf{V}$ to the set of value types. We use the following notation:

- $\mathsf{dom}(\Gamma)$ denotes $\{u \mid u\!:\!\tau \in \Gamma\}$ and Γ/A denotes $\{u\!:\!\tau \in \Gamma \mid u \notin A\}$.
- $\Gamma, u\!:\!\tau$ means $\Gamma \cup \{u\!:\!\tau\}$, together with the assumption $u \notin \mathsf{dom}(\Gamma)$.
- $\Delta \leq \Gamma$ means $\forall\, u \in \mathsf{dom}(\Gamma),\ \Delta(u) \leq \Gamma(u)$.

The partial meet $\sqcap$ and partial join $\sqcup$, defined on types, are generalised to type environments in the standard manner:

- $\Gamma \sqcap \Delta \stackrel{\mathrm{def}}{=} \Gamma/\mathsf{dom}(\Delta) \cup \Delta/\mathsf{dom}(\Gamma) \cup \{u\!:\!(\Delta(u) \sqcap \Gamma(u)) \mid u \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)\}$
- $\Gamma \sqcup \Delta \stackrel{\mathrm{def}}{=} \{u\!:\!(\Delta(u) \sqcup \Gamma(u)) \mid u \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)\}$

One can easily check that they satisfy the properties of partial meets and partial joins respectively.

Typing Assignments are formulas $P : \rho$ for any term P and any type ρ. We write $\Gamma \vdash P : \rho$ if the formula $P : \rho$ is provable from a typing environment Γ using the Standard Typing System given in Figure 4. This is divided in two parts. The first is inherited from the λ-calculus, while the second is a simple adaptation of the IO-typing system from [27, 15].

EXAMPLE 2.4. (Typed sq server) We may now revisit the example discussed above, assigning appropriate types to the channel names and variables involved. In the definition of $\mathbf{sq}(a)$ a pair of values are input, a natural number and a channel respectively, and this channel will be used to transmit a natural number. So the following annotation would be reasonable:

$$\mathbf{sq}(a) \Longleftarrow\ *a?(y\!:\!\mathtt{int}, z\!:\!(\mathtt{int})^{\mathtt{IO}}).\ z!\langle \mathtt{sq}(y)\rangle$$

However with this typing a user of this process, when transmitting to it a return channel, is also giving the process permission to receive on that channel. To provide protection against possible misuse a more appropriate type annotation would be

$$\mathbf{sq}(a) \Longleftarrow\ *a?(y\!:\!\mathtt{int}, z\!:\!(\mathtt{int})^{\mathtt{O}}).\ z!\langle \mathtt{sq}(y)\rangle$$

where the process only receives the output capability on the return channel. Now we have $\Gamma \vdash \mathbf{sq}(a) : \mathtt{proc}$ for any typing function Γ such that $\Gamma(a) \leq (\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{I}}$. Then by ABS in Figure 4, we have:

$$\vdash \quad \lambda(x\!:\!(\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{I}}).\,\mathbf{sq}(x) : (\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{I}} \to \mathtt{proc}$$

Common Typing Rules:

$$\textsc{Id}: \quad \Gamma, u\!:\!\tau \vdash u : \tau \qquad \textsc{Sub}: \quad \frac{\Gamma \vdash P : \rho \quad \rho \leq \rho'}{\Gamma \vdash P : \rho'}$$

Functional Typing Rules:

$$\textsc{Const}: \quad \Gamma \vdash 1 : \mathtt{nat} \quad \text{etc.} \qquad \textsc{Abs}: \quad \frac{\Gamma, x\!:\!\tau \vdash P : \rho}{\Gamma \vdash \lambda(x\!:\!\tau).P : \tau \to \rho}$$

$$\textsc{App}: \quad \frac{\Gamma \vdash P : \tau \to \rho \quad \Gamma \vdash Q : \tau}{\Gamma \vdash PQ : \rho}$$

Process Typing Rules:

$$\textsc{In}: \quad \frac{\Gamma \vdash u : (\tilde{\tau})^{\mathtt{I}} \quad \Gamma, \tilde{x}\!:\!\tilde{\tau} \vdash P : \mathtt{proc}}{\Gamma \vdash u?(\tilde{x}\!:\!\tilde{\tau}).P : \mathtt{proc}} \qquad \textsc{Nil}: \quad \Gamma \vdash \mathbf{0} : \mathtt{proc}$$

$$\textsc{Out}: \quad \frac{\Gamma \vdash u : (\tilde{\tau})^{\mathtt{O}} \quad \Gamma \vdash V_i : \tau_i \quad \Gamma \vdash P : \mathtt{proc}}{\Gamma \vdash u!\langle \tilde{V} \rangle P : \mathtt{proc}} \qquad \textsc{Rep}: \quad \frac{\Gamma \vdash P : \mathtt{proc}}{\Gamma \vdash *P : \mathtt{proc}}$$

$$\textsc{Res}: \quad \frac{\Gamma, a\!:\!\sigma \vdash P : \mathtt{proc}}{\Gamma \vdash (\nu a\!:\!\sigma)P : \mathtt{proc}} \qquad \textsc{Par}: \quad \frac{\Gamma \vdash P : \mathtt{proc} \quad \Gamma \vdash Q : \mathtt{proc}}{\Gamma \vdash P \,|\, Q : \mathtt{proc}}$$

FIGURE 4. Standard Typing System for $\pi\lambda$

which means that should x be instantiated by a channel whose capability is *dominated by* $(\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{I}}$, then it becomes a safe process. □

EXAMPLE 2.5. (comparison with [27]) Our general form of IO-types, where input and output capabilities on a channel may be different [16, 15], gives us more typable terms than [27], which immediately arise when we integrate with the subtyping of the functional types. Suppose $\mathtt{nat} \leq \mathtt{real}$,[2] a literal succ for successing natural numbers which has a type $\mathtt{nat} \to \mathtt{nat}$, and a literal sq for squaring real numbers which has a type $\mathtt{real} \to \mathtt{real}$.

$$\begin{aligned} P &\stackrel{\text{def}}{=} a!\langle b\rangle \,|\, b?(x\!:\!\mathtt{nat}).\, c!\langle \mathtt{succ}(x)\rangle \,|\, b!\langle 1\rangle \\ Q &\stackrel{\text{def}}{=} a!\langle d\rangle \,|\, d?(x\!:\!\mathtt{real}).\, e!\langle \mathtt{sq}(x)\rangle \,|\, d!\langle 1.1\rangle \\ R &\stackrel{\text{def}}{=} a?(x).\, (x?(y\!:\!\mathtt{real}).\, e!\langle \mathtt{sq}(x)\rangle \,|\, x!\langle 1\rangle) \end{aligned}$$

Then we can find both interactions between P and R, and Q and R, do not disturb the manner of the value constraints. In our typing system, $P \,|\, Q \,|\, R$ is typable in

[2] All theorems of this paper are extended to the atomic subtyping system [20] which includes $\mathtt{nat} \leq \mathtt{real}$.

the following environment.

$$\Gamma = a\!:\!(\langle(\mathtt{real}),(\mathtt{nat})\rangle)^{\mathtt{IO}}, b\!:\!(\mathtt{nat})^{\mathtt{IO}},\ c\!:\!(\mathtt{nat})^{\mathtt{O}},\ d\!:\!(\mathtt{real})^{\mathtt{IO}},\ e\!:\!(\mathtt{real})^{\mathtt{O}}$$

But it is impossible to assign any type to a based on the ordering of [27]. If we assign $((\mathtt{nat})^{\mathtt{IO}})^{\mathtt{IO}}$ or $((\mathtt{real})^{\mathtt{IO}})^{\mathtt{IO}}$ which are types in [27] to a, processes behave inconsistently. Consider the following process in which a has a type $((\mathtt{nat})^{\mathtt{IO}})^{\mathtt{IO}}$.

$$R' \stackrel{\text{def}}{=} a?(x).\, (x?(y\!:\!\mathtt{nat}).\, c!\langle \mathtt{succ}(x)\rangle \,|\, x!\langle 1\rangle)$$

The reader can easily check the communication between R' and Q violates the value usage. Note the similar example can be given only in the π-terms without adding any functional expressions (simply replace, e.g. nat with $()^{\mathtt{IO}}$ and real with $()^{\mathtt{O}}$, respectively). Hence our subtyping ordering based on [15] is not only natural to integrate with functional subtypes, but also *strictly more general* than that of [27] even in the pure polyadic π-calculus. □

This standard typing system satisfies the usual requirements of a reasonable typing system, such as Weakening and Type Specialisation, together with the following subject reduction theorem.

THEOREM 2.6. (Subject Reduction)

If $\Gamma \vdash P : \rho$ *and* $P \longmapsto\!\!\!\!\!\rightarrow P'$, *then we have* $\Gamma \vdash P' : \rho$.

Proof See Appendix A. □

3 Locality of Channels in Distributed Higher Order π-calculus

In this section we first extend the language by introducing an explicit, but simple, representation of distribution of processes. Then we discuss the main topic of the paper, difficulty to ensure locality of names in D$\pi\lambda$.

DISTRIBUTED HIGHER ORDER π-CALCULUS The extended syntax for distributed processes is given by in Figure 5. Intuitively $N \,\|\, M$ represents two systems N, M running at two physically distinct locations, while the process $\mathtt{Spawn}(P)$ creates a new location at which the process P is launched. A more comprehensive representation of distribution could be given, as in [6, 15, 34], by associating names with locations and allowing these names to be generated dynamically and transmitted between processes. However the simple syntax given above is sufficient for our purposes: the development of a static typing system which guarantees the locality of channels in a distributed setting. We believe that our results will be adaptable without difficulty to more complicated languages.

The reduction semantics of the previous section is extended to the new language, D$\pi\lambda$, in a straightforward manner, by defining a reduction relation $N \longrightarrow N'$ between system; we use $N \longrightarrow\!\!\!\!\!\rightarrow N'$ to denote the corresponding multi-step

Syntax:

System: $M, N, \ldots \in \mathit{System} \quad ::= \quad P \mid N \| M \mid (\nu a{:}\sigma)N$

Term: $P, Q, \ldots \in \mathit{Term} \quad ::= \quad \mathtt{Spawn}(P) \mid \cdots$ as Figure 1

Value: as Figure 1

Distributed Reduction Rules:

(process) $\dfrac{P \longmapsto P'}{P \longrightarrow P'}$ $\quad P \longmapsto P'$ from Figure 2

(spawn) $(\cdots Q \mid \mathtt{Spawn}(P)) \longrightarrow (\cdots Q) \| P$

(com$_s$) $(u?(\tilde{x}{:}\tilde{\tau}).P \mid \cdots) \| (u!\langle\tilde{V}\rangle Q \mid \cdots) \longrightarrow (P\{\tilde{V}/\tilde{x}\} \mid \cdots) \| (Q \mid \cdots)$

(par$_s$) $\dfrac{M \longrightarrow M'}{M \| N \longrightarrow M' \| N}$ (res$_s$) $\dfrac{N \longrightarrow N'}{(\nu a{:}\sigma)N \longrightarrow (\nu a{:}\sigma)N'}$ (str$_s$) $\dfrac{N \equiv N' \longrightarrow M' \equiv M}{N \longrightarrow M}$

FIGURE 5. Syntax and Distributed Reduction in D$\pi\lambda$

relation. The definition is outlined in Figure 5 and uses a structural equivalence on systems, defined by changing "$|$" to "$\|$" and P, Q, R to M, N, N' in Figure 2. The first two rules are the most important, namely spawning of a process at a new location (spawn) and communication between physically distinct locations, (com$_s$).

DEFINING LOCALITY We require that every input channel name is associated with a unique location. This is violated in, for example,

$$a?(y).\,P \;\;\|\;\; (a?(z).\,Q \mid b?(x_1).\,R_1 \mid b?(x_2).\,R_2)$$

because the name a can receive input at two distinct locations. Note however that the name b is located uniquely, although at that location a call can be serviced in two different ways.

A formal definition of this concept (or rather its complement), *locality error*, is given in Figure 6, using a predicate on systems, $N \xrightarrow{lerr}$. Intuitively this should be read as saying: in the system N there is a runtime error, namely there is some name a which is ready to receive input at two distinct locations. The definition is by a straightforward structural induction on systems and uses an auxiliary predicate $P \downarrow a^{\mathtt{I}}$ which is satisfied when P can immediately perform input on name a. Formally we would like to develop a type system such that for the system N, being well-typed ensures $N \not\xrightarrow{lerr}$.

Input Predicate:

$$a?(\tilde{x}).\,P \downarrow a^{\mathtt{I}} \qquad \frac{P \downarrow a^{\mathtt{I}}}{(P \mid Q) \downarrow a^{\mathtt{I}}} \qquad \frac{Q \downarrow a^{\mathtt{I}}}{(P \mid Q) \downarrow a^{\mathtt{I}}} \qquad \frac{P \downarrow a^{\mathtt{I}} \quad a \neq b}{(\nu b)P \downarrow a^{\mathtt{I}}} \qquad \frac{P \downarrow a^{\mathtt{I}}}{*P \downarrow a^{\mathtt{I}}}$$

$$\frac{M \downarrow a^{\mathtt{I}}}{(N \| M) \downarrow a^{\mathtt{I}}} \qquad \frac{N \downarrow a^{\mathtt{I}}}{(N \| M) \downarrow a^{\mathtt{I}}} \qquad \frac{N \downarrow a^{\mathtt{I}} \quad a \neq c}{(\nu c)N \downarrow a^{\mathtt{I}}}$$

Locality Error:

$$\frac{N \downarrow a^{\mathtt{I}} \quad M \downarrow a^{\mathtt{I}}}{(N \| M) \xrightarrow{lerr}} \qquad \frac{N \xrightarrow{lerr}}{(N \| M) \xrightarrow{lerr}} \qquad \frac{N \xrightarrow{lerr}}{(M \| N) \xrightarrow{lerr}} \qquad \frac{N \xrightarrow{lerr}}{(\nu c)N \xrightarrow{lerr}}$$

FIGURE 6. Locality Error

Such a typing system is given by the Distributed Typing Rules in Figure 7, where the judgements take the form $\Gamma \vdash_1 N$, with Γ being again a typing environment. The most essential rule is PAR$_l$; this says that $N_1 \| N_2$ is typable with respect to an environment Δ if Δ can be written as $\Gamma_1 \sqcap \Gamma_2$, where Γ_1 and Γ_2 are *system composable*, $\Gamma_1 \asymp_1 \Gamma_2$ and N_i is typable with respect to Γ_i.

DEFINITION 3.1. Γ_1 and Γ_2 are *composable*, written by $\Gamma_1 \asymp \Gamma_2$, if $\Gamma_1 \sqcap \Gamma_2$ is defined, and Γ_1 and Γ_2 are *system-composable*, written by $\Gamma_1 \asymp_1 \Gamma_2$, if $\Gamma_1 \asymp \Gamma_2$ and $u{:}\langle S_{i\mathtt{I}}, S_{i\mathtt{O}}\rangle \in \Gamma_i$ $(i = 1, 2)$ implies $S_{1\mathtt{I}} = \top$ or $S_{2\mathtt{I}} = \top$. □

Now let us say a channel type σ is *local* if σ has an input capability, i.e $\sigma = \langle(\tilde{\tau}), S_{\mathtt{O}}\rangle$. We also call a channel u is *local under* Γ if $\Gamma(u)$ is local. Then intuitively Γ_1 and Γ_2 being system composable means that if a channel a is local in Γ_1 it must not be local in the other environment Γ_2, and vice-versa.

To prove *Type Safety* for the typing system, that is eliminates locality errors, we first require a correspondence between the input capability and the input predicate:

LEMMA 3.2.

(1) (process) $P \downarrow a^{\mathtt{I}}$ and $\Gamma \vdash P : \mathtt{proc}$ imply $\Gamma \vdash a : (\tilde{\tau})^{\mathtt{I}}$ for some $\tilde{\tau}$.
(2) (system) $N \downarrow a^{\mathtt{I}}$ and $\Gamma \vdash_1 N$ imply $\Gamma \vdash a : (\tilde{\tau})^{\mathtt{I}}$ for some $\tilde{\tau}$.

Proof An easy by induction on the proof of $P \downarrow a^{\mathtt{I}}$ and $N \downarrow a^{\mathtt{I}}$. □

THEOREM 3.3. (Type Safety) $\Gamma \vdash_1 N$ *implies* $N \not\xrightarrow{lerr}$.

Proof By induction on derivation of $\Gamma \vdash_1 N$.

- $\Gamma \vdash_1 N$ is inferred by $\Gamma \vdash_1 P \equiv N$. Then it is vacuous since $P \not\xrightarrow{lerr}$.

Local Distributed Rules:

$$\text{SPAWN:}\quad \frac{\Gamma \vdash P : \mathtt{proc}}{\Gamma \vdash \mathtt{Spawn}(P) : \mathtt{proc}}$$

$$\text{INTRO:}\ \frac{\Gamma \vdash P : \mathtt{proc}}{\Gamma \vdash_1 P} \qquad \text{PAR}_l\text{:}\ \frac{\Gamma \vdash_1 N \quad \Delta \vdash_1 M \quad \Gamma \asymp_1 \Delta}{\Gamma \sqcap \Delta \vdash_1 N \,\|\, M} \qquad \text{RES}_l\text{:}\ \frac{\Gamma, a : \sigma \vdash_1 M}{\Gamma \vdash_1 (\nu a : \sigma) M}$$

FIGURE 7. Local Distributed Typing Rules

- $\Gamma \vdash_1 (\nu a : \sigma) N$ because $\Gamma, a : \sigma \vdash_1 N$. Then by induction, we have $N \overset{lerr}{\not\longrightarrow}$. Hence we have $(\nu a : \sigma) N \overset{lerr}{\not\longrightarrow}$.
- $\Gamma_1 \sqcap \Gamma_2 \vdash_1 M \,\|\, N$ is inferred by $\Gamma_1 \vdash_1 M$ and $\Gamma_2 \vdash_1 N$ with $\Gamma_1 \asymp_1 \Gamma_2$. By induction, we know $M \overset{lerr}{\not\longrightarrow}$ and $N \overset{lerr}{\not\longrightarrow}$. So the only possibility to infer $M \,\|\, N \overset{lerr}{\longrightarrow}$ is to use the first rule. We show that is impossible. Suppose $M \downarrow a^{\mathtt{I}}$. By Lemma 3.2, we have $\Gamma_1 \vdash_1 a : (\tilde{\tau})^{\mathtt{I}}$. Since $\Gamma_1 \asymp_1 \Gamma_2$, this means $\Gamma_2(a) = \langle \top, S \rangle$ for some S. Hence $\Gamma_1 \not\vdash a : (\tilde{\tau}')^{\mathtt{I}}$ for any $\tilde{\tau}'$. Applying the lemma again, we have $M \not\downarrow a^{\mathtt{I}}$. $\square$

However it is easy to see that typability, as defined above, is not closed under reduction: $N \overset{lerr}{\not\longrightarrow}$ and $N \longrightarrow\!\!\!\!\rightarrow N'$ does *not* imply $N' \overset{lerr}{\not\longrightarrow}$. Various counter-examples will be given in the next subsection.

DIFFICULTIES IN PRESERVING LOCALITY IN D$\pi\lambda$ There are basically two reasons why locality is not preserved after communication. The first reason is the use of a name received from another location as an input subject. The second, which is more complicated, concerns the parameterisations of processes and the instantiation of variables which occur in outgoing values.

We first start with a simple example which does not involve process passing.

EXAMPLE 3.4. Consider the system

$$a?(x).\,P \,|\, b!\langle a \rangle \;\|\; b?(y).\, y?(z).\, Q$$

Then it is easy to check that this can be typed with PAR_l in Figure 6. However after one reduction step, the communication along b, we obtain (up to structural equivalence)

$$a?(x).\,P \;\|\; a?(z).\,Q$$

which is no longer typable.

The problem in the first system occurs because, although a is already located, an *input capability* on a is also transmitted along b to another location; in any typing Γ of the system we must have $\Gamma(b) \leq (\sigma)^{\mathtt{I}}$, for some σ.

This misbehaviour can be eliminated by revising the typing system for processes, Figure 4, so as to restrict the transmission of such capabilities. Indeed the typing system which enforces locality in the π-calculus given in [3, 22] can be viewed in this manner. However the presence of higher-order processes makes the situation more complicated, as the following example shows.

EXAMPLE 3.5. Let V denote the value $\lambda().\,\mathbf{sq}(a)$ in the slightly modified system

$$a?(x).\,P \,|\, b!\langle V \rangle \;\|\; b?(Y).\,Y\,()$$

Once more this is a typable configuration; nevertheless, after the transmission of the value V to the new site and a reduction we get a system which violates our locality conditions.

Next consider a similar code where V denotes $\lambda(x).\,\mathbf{sq}(x)$.

$$a?(x).\,P \,|\, b!\langle V \rangle \;\|\; b?(Y).\,(Y\,c) \,|\, c?(x).\,Q$$

Then this does not destroy the locality conditions. $\square$

Certain values are *sendable* in that their transfer from location to location will never lead to a locality error. For example, the first value $\lambda().\,\mathbf{sq}(a)$ is immediately not sendable, although $\lambda(x).\,\mathbf{sq}(x)$ will be sendable, because it contains no free occurrence of input channels. However the algebra of *sendable* and non-*sendable* terms is not straightforward.

EXAMPLE 3.6. Let V be a seemingly sendable value $\lambda(x).\,\mathbf{sq}(x)$ in the system

$$d?(X).\,X() \,|\, b!\langle V \rangle \;\|\; b?(Y).\,d!\langle \lambda().(Y\,c) \rangle \,|\, c?(x).\,Q$$

Here V is transmitted along b across locations, where it is used to construct a new value, $\lambda().(V\,c)$; this is then transmitted across locations via d and when it is run we obtain once more a locality error.

More interestingly, the following does not disturb locality although we pass $\lambda().\,\mathbf{sq}(c)$ directly:

$$d?(X).\,X() \;\|\; b!\langle \lambda().\,\mathbf{sq}(c) \rangle \,|\, b?(Y).\,Y\,() \,|\, c?(x).\,Q$$

However, the following does violate the locality conditions.

$$d?(X).\,X() \;\|\; b!\langle \lambda().\,\mathbf{sq}(c) \rangle \,|\, b?(Y).\,d!\langle Y \rangle \,|\, c?(x).\,Q \qquad \square$$

Again the problem in the first example is the non-sendable values $\lambda().(V\,c)$, which does not appear in the original system, but constructed dynamically. Similarly in the third example, only Y appears an object of $d!\langle Y \rangle$, but it was dynamically instantiated by non-sendable value.

We need a new set of types which includes *sendable*/non-*sendable* types and a typing system which controls the formation of values and ensures that in every

Types:

Term Type:	ρ	$::=$	$\pi \mid \tau$
Process Type:	π	$::=$	$\texttt{proc} \mid \mathbf{s}(\texttt{proc})$
Value Type:	τ	$::=$	$\texttt{unit} \mid \texttt{nat} \mid \texttt{bool} \mid \sigma \mid \mathbf{s}(\tau) \quad \text{with } \tau \neq \sigma$
		$\mid$	$\tau \to \rho \quad \text{with } \rho \leq \mathbf{s}(\rho') \Rightarrow \tau \leq \mathbf{s}(\tau')$
Channel Type:	σ	$::=$	$\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$ with $S_{\mathtt{I}} \geq S_{\mathtt{O}},\ S_{\mathtt{I}} \neq \bot$ and $S_{\mathtt{O}} \neq \top$
		$\mid$	$\mathbf{s}(\langle \top, S_{\mathtt{O}} \rangle)$
Sort Type:	S	$::=$	as in Figure 3.

Ordering: All rules from Figure 3 and

(trans)	$\rho_1 \leq \rho_2 \quad \rho_2 \leq \rho_3 \ \Rightarrow\ \rho_1 \leq \rho_3$
(mono)	$\rho \leq \rho' \ \Rightarrow\ \mathbf{s}(\rho) \leq \mathbf{s}(\rho')$
(sendable)	$\mathbf{s}(\rho) \leq \rho$
(id)	$\mathbf{s}(\rho) \leq \mathbf{s}(\mathbf{s}(\rho))$
(lift)	$\mathbf{s}(\tau) \to \mathbf{s}(\rho) \leq \mathbf{s}(\mathbf{s}(\tau) \to \mathbf{s}(\rho))$

FIGURE 8. Locality types for $\mathrm{D}\pi\lambda$

occurrence of $b!\langle V \rangle$, where the term V can be exported to another location, it can only evaluate to a value of *sendable* type.

4 Type Inference System for Locality

This section formalises a new typing system for processes. The important point of our typing is that if each process in each location is statically type-checked, we can automatically ensure that, in the global environment, input capability always resides at a unique location even after arbitrary computation.

LOCAL TYPING SYSTEM We add a new type constructor $\mathbf{s}(\rho)$ for sendable terms; the formation rules and ordering are given in Figure 8. The side condition of arrow types simply avoids, as we will see, a sendable term having a non-sendable subterm; e.g. if either P or Q is non-sendable, then $P\,Q$ will automatically be non-sendable. A similar side condition on arrow types can be found in the *passive types* in [26].

The first rule ensures that $\leq$ is a preorder. The second extra ordering says that the constructor $\mathbf{s}(\ )$ preserves subtyping, and the third that all sendable values are also values. In conjunction with (id), this rule implies that sendability is idempotent, $\mathbf{s}(\mathbf{s}(\rho)) \simeq \mathbf{s}(\rho)$ with $\simeq \stackrel{\text{def}}{=} \leq \cap \geq$. Similarly with (lift), we have: $\mathbf{s}(\mathbf{s}(\tau) \to \mathbf{s}(\rho)) \simeq \mathbf{s}(\tau) \to \mathbf{s}(\rho)$. Indeed this can be generalised:

LEMMA 4.1. *For any $k \geq 0$, we have:*

$$\mathbf{s}(\mathbf{s}(\tau_1) \to \cdots \to \mathbf{s}(\tau_k) \to \mathbf{s}(\rho)) \quad \simeq \quad \mathbf{s}(\tau_1) \to \cdots \to \mathbf{s}(\tau_k) \to \mathbf{s}(\rho) \ .$$

Proof A simple induction on k. □

We will change the distributed typing system by ensuring a value V will only be exportable to another location if it can be assigned a type of the form $\mathbf{s}(\tau)$. However because our typing system has a subsumption rule a more general statement would be that to be exportable we assign to V a type τ' such that $\tau' \leq \mathbf{s}(\tau)$, for some τ. This leads to a formal definition of *sendable types*.

DEFINITION 4.2. Let Sble, the set of sendable types, be the least set of types which satisfies:

- $\mathbf{s}(\rho) \in \mathsf{Sble}$ for any type $\mathbf{s}(\rho)$.
- $\tau,\ \rho \in \mathsf{Sble}$ implies $\tau \to \rho \in \mathsf{Sble}$.

We say ρ is *sendable* if $\rho \in \mathsf{Sble}$. □

The main properties of the set of sendable types is given in the following proposition:

PROPOSITION 4.3.

1. Sble *is downwards closed with respect to subtyping:* $\rho' \leq \rho$ *and* $\rho \in \mathsf{Sble}$ *implies* $\rho' \in \mathsf{Sble}$
2. $\rho \in \mathsf{Sble}$ *if and only if* $\rho \simeq \mathbf{s}(\rho)$
3. $\rho \in \mathsf{Sble}$ *if and only if* $\rho \leq \mathbf{s}(\rho')$ *for some* ρ'.

Proof

1. The proof is by structural induction on ρ'; so we can assume that the statement is true of all ρ'' which are structurally less than ρ'. We now do a further induction on the proof of $\rho' \leq \rho$. Most of the cases are straightforward; for example the use of any of the axioms in Figure 8 are immediate. The only non-trivial case is when ρ', ρ have the structure $\tau_1 \to \rho_1, \tau_2 \to \rho_2$, respectively, where $\tau_2 \leq \tau_1$ and $\rho_1 \leq \rho_2$. We know $\rho_2 \in \mathsf{Sble}$ and therefore by induction $\rho_1 \in \mathsf{Sble}$. Also because of the constraint on the formation of arrow types we know that $\tau_1 \leq \mathbf{s}(\tau_3)$, for some τ_3. By construction $\mathbf{s}(\tau_3) \in \mathsf{Sble}$ and therefore, again by induction $\tau_1 \in \mathsf{Sble}$. It follows that $\tau_1 \to \rho_1 \in \mathsf{Sble}$.
2. Suppose $\rho \in \mathsf{Sble}$. A simple proof by induction on why $\rho \in \mathsf{Sble}$ gives $\rho \simeq \mathbf{s}(\rho)$, remembering that for any type ρ' we have $\mathbf{s}(\rho') \leq \rho'$ (if $\mathbf{s}(\rho')$ is defined). The case when ρ is an arrow type uses Lemma 4.1. The converse follows immediately from part one.
3. Now follows from parts one and two. □

The last statement of this Proposition is particularly relevant; in our revised typing system a value can only be exported to a new site if it can be assigned a type in Sble. Note also that $\tau \to \rho \in \mathsf{Sble}$ if and only if both $\tau \in \mathsf{Sble}$ and $\rho \in \mathsf{Sble}$; in other words $\tau \to \rho$ is sendable only if both τ and ρ are sendable. We can also restate the condition on the formation on arrow types in Figure 8 as saying that if ρ is sendable then $\tau \to \rho$ is a valid type only if τ is sendable. However note that $\mathbf{s}(\tau \to \rho)$ is sendable even if τ and ρ are not sendable.

At first sight it may be natural to allow types such as $\langle \top, S_{\mathtt{O}} \rangle$ to also be in Sble. However if this were allowed, Sble is no longer be downward closed with respect to $\leq$, a property required in the proof of Lemma 4.12 (2), in turn a technical result crucial in our proof of the Subject Reduction Theorem.

We can also state a normal form theorem for valid types:

LEMMA 4.4. (Normal form) *For every valid type ρ there exists some $k \geq 0$ such that one of the following holds:*

(1) $\rho \simeq \mathbf{s}(\tau_1) \to \cdots \to \mathbf{s}(\tau_k) \to \mathbf{s}(\rho_1)$,
(2) $\rho \simeq \tau_1 \to \cdots \to \tau_k \to \rho_G$,
(3) $\tau \simeq \mathbf{s}(\tau_1 \to \cdots \to \tau_k \to \rho_G)$

where ρ_1 takes the form of (2) and ρ_G is either $\mathtt{proc}$, $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$ *or a base type.*

Proof By structural induction on τ, using Lemma 4.1 and Proposition 4.3. □

The essential order theoretic property on the subtyping relation is also preserved in this extension.

PROPOSITION 4.5. (Ordering) *The subtyping relation over types defined in Figure 8 is finite bounded complete,* FBC.

Proof We first extend the definition of $\sqcap$ and $\sqcup$ in § 2.3 to the sendable types as follows.

- $\mathbf{s}(\rho_1) \sqcap \mathbf{s}(\rho_2) = \mathbf{s}(\rho_1 \sqcap \rho_2)$ and $\mathbf{s}(\rho_1) \sqcup \mathbf{s}(\rho_2) = \mathbf{s}(\rho_1 \sqcup \rho_2)$
- $\mathbf{s}(\rho_1) \sqcap \rho_2 = \mathbf{s}(\rho_1 \sqcap \rho_2)$ and $\mathbf{s}(\rho_1) \sqcup \rho_2 = \rho_1 \sqcup \rho_2$ with $\rho_2 \neq \mathbf{s}(\rho_2')$

The non-trivial cases in the proof that $\sqcap$ and $\sqcup$ are indeed partial glb and lub operators are arrow types and channel types. We use Proposition 4.3, together with Lemma 4.4. See Appendix B.1 for the proofs. □

The new type inference system, with judgements of the form $\Gamma \vdash_1 P : \rho$, is given in Figure 9 and uses the notion of *sendable type environments*, environments which only use sendable types.

DEFINITION 4.6. A typing environment Δ is *sendable*, written $\Delta \vdash_1 \mathtt{SBL}$, if $u{:}\tau \in \Delta$ implies $\tau \in \mathsf{Sble}$ or $a{:}\sigma \in \Delta$ implies $\sigma \in \langle \top, S_{\mathtt{O}} \rangle$. □

Send Rules:

$$\textsc{Const}_l:\ \frac{\Gamma \vdash_1 l : \mathtt{nat}}{\Gamma \vdash_1 l : \mathbf{s}(\mathtt{nat})}\ \text{etc.} \qquad \textsc{Chan}_l:\ \frac{\Gamma \vdash_1 a : \langle \top, S \rangle}{\Gamma \vdash_1 a : \mathbf{s}(\langle \top, S \rangle)}$$

$$\textsc{Term}_l:\ \frac{\Delta \vdash_1 P : \rho \quad \Delta \vdash_1 \mathtt{SBL} \quad \Delta \geq \Gamma}{\Gamma \vdash_1 P : \mathbf{s}(\rho)} \qquad \textsc{Spawn}_l:\ \frac{\Gamma \vdash_1 P : \mathbf{s}(\mathtt{proc})}{\Gamma \vdash_1 \mathtt{Spawn}(P) : \mathbf{s}(\mathtt{proc})}$$

Common Rules: as in Figure 4.

Functional Rules: as in Figure 4.

Process Rules:

$$\textsc{Out}_d:\ \frac{\begin{array}{c}\Gamma \vdash_1 u : (\mathbf{s}(\tilde{\tau}))^{\mathtt{O}} \\ \Gamma \vdash_1 V_i : \mathbf{s}(\tau_i) \quad \Gamma \vdash_1 P : \pi\end{array}}{\Gamma \vdash_1 u!\langle \tilde{V} \rangle P : \pi} \qquad \textsc{Out}_l:\ \frac{\begin{array}{c}\Gamma \vdash_1 u : \langle (\tilde{\tau}'), (\tilde{\tau}) \rangle \\ \Gamma \vdash_1 V_i : \tau_i \quad \Gamma \vdash_1 P : \mathtt{proc}\end{array}}{\Gamma \vdash_1 u!\langle \tilde{V} \rangle P : \mathtt{proc}}$$

NIL,REP,PAR,RES as in Figure 4 with $\mathtt{proc}$ replaced by π, and IN the same as in Figure 4.

Local Distributed Rules: $\textsc{Par}_l$ and $\textsc{Res}_l$ as in Figure 7 and INTRO as in Figure 7 with $\vdash$ replaced by $\vdash_1$ in INTRO.

FIGURE 9. Locality Typing System for D$\pi\lambda$

In Figure 9 the **Send Rules** determine which values can be exported to other locations, either by spawning or by communication. All constants and output capabilities on channels are automatically sendable. The crucial rule is $\textsc{Term}_l$, which says that in a general term is sendable only if it can be derived from a sendable type environment.

The rules for processes also require minor modifications. We can create a process by spawn if it is sendable. In $\textsc{Out}_d$ we require that values which will be sent across locations to have sendable types. However if the transmission is only done in the same location, this condition is relaxed; in $\textsc{Out}_l$, the message is guaranteed to be transmitted within the same location since name a has an input capability. Note also an input process has always the non-sendable type $\mathtt{proc}$.

EXAMPLE 4.7. (Sq-server) In the following, we offer a non-trivial example of the use of sendability in typing. Recall Examples 2.1 and 2.4, and let us define

$$\sigma = (\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{I}} \qquad \tau = \sigma \to \mathtt{proc} \qquad \sigma' = (\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{IO}}$$

First we note $\lambda(x{:}\sigma).\ \mathbf{sq}(x)$ has a sendable type $\mathbf{s}(\sigma \to \mathtt{proc})$; the derivation is similar to that in Example 2.4, followed by an application of the rule $\textsc{Term}_l$.

Then **SqServ** is typed as follows:

$$\mathrm{req}:((\tau)^{\mathtt{O}})^{\mathtt{I}} \vdash_1 *\mathrm{req}?(r:(\tau)^{\mathtt{O}}).r!\langle\lambda(x:\sigma).\,\mathbf{sq}(x)\rangle : \mathtt{proc}$$

Here the type declaration "$(\tau)^{\mathtt{O}}$" of r ensures that **SqServ** does not create a new input subject by a value received though channel "req".

Next for **Client**, first let us define its body as $P \equiv (\,X a \mid a!\langle 1, c_1\rangle \mid \cdots)$. To accept $\lambda(x:\sigma).\,\mathbf{sq}(x)$ from the server and create $\mathbf{sq}(a)$ by applying a to $\lambda(x:\sigma).\,\mathbf{sq}(x)$, a will be used with both input/output capabilities in P. Hence P is typed as: $X:\tau,\ a:\sigma' \vdash_1 P:\mathtt{proc}$. Now let us define $\Gamma = \mathrm{req}:((\tau)^{\mathtt{O}})^{\mathtt{O}}, r:(\tau)^{\mathtt{IO}}$. Since $(\tau)^{\mathtt{IO}} \leq (\tau)^{\mathtt{I}}$, by applying RES and IN, we have:

$$\Gamma \vdash_1 r?(X:\tau).\,(\nu a:\sigma')P:\mathtt{proc}$$

To output r through "req", r should have a sendable type. Then, by $(\tau)^{\mathtt{IO}} \leq (\tau)^{\mathtt{O}}$ and CHAN$_l$ rule, we have:

$$\frac{\Gamma \vdash_1 r:(\tau)^{\mathtt{O}}}{\Gamma \vdash_1 r:\mathbf{s}((\tau)^{\mathtt{O}})}$$

The type of the channel "req" in the client is inferred by $\mathbf{s}((\tau)^{\mathtt{O}}) \leq (\tau)^{\mathtt{O}}$ as well as the contravariance of output capability:

$$\Gamma \vdash_1 \mathrm{req}:(\mathbf{s}((\tau)^{\mathtt{O}}))^{\mathtt{O}}$$

Combining these three, we now infer:

$$\mathrm{req}:((\tau)^{\mathtt{O}})^{\mathtt{O}} \vdash_1 (\nu r:(\tau)^{\mathtt{IO}})\mathrm{req}!\langle r\rangle r?(X:\tau).\,(\nu a:\sigma')P \equiv \textbf{Client}:\mathtt{proc}$$

Finally since $\{\mathrm{req}:((\tau)^{\mathtt{O}})^{\mathtt{I}}\} \asymp_1 \{\mathrm{req}:((\tau)^{\mathtt{O}})^{\mathtt{O}}\}$, both systems are system composable.

$$\mathrm{req}:((\tau)^{\mathtt{O}})^{\mathtt{IO}} \vdash_1 \textbf{SqServ} \parallel \textbf{Client}$$

Observe that:

(1) The sendable type $\mathbf{s}(\sigma \to \mathtt{proc})$ of $\lambda(x:\sigma).\,\mathbf{sq}(x)$ makes it possible to create a new server $\mathbf{sq}(a)$ in the client side.

(2) r is declared with both input and output capabilities in the **Client**. The **Client** itself uses the input capability but, because of the type of "req" it only sends the output capability to **SqServ**. This form of communication is essential to represent a continuation passing style programming in the π-calculus as studied in [17, 27, 31, 32]. Moreover it demonstrates the need for non-trivial subtyping on channels. □

We now give a series of examples which show that our typing systems eliminates various forms of behaviour which destroy our locality conditions.

EXAMPLE 4.8. Recall the first system in Example 3.4.

$$a?(x).\,P \mid b!\langle a\rangle \parallel b?(y).\,y?(z).\,Q$$

Suppose $\Gamma_1 \vdash a?(x).\,P \mid b!\langle a\rangle$ and $\Gamma_2 \vdash_1 b?(y).\,y?(z).\,Q$. Then we have $\Gamma_2(b) \leq ((\sigma)^{\mathtt{I}})^{\mathtt{I}}$ for some σ. Since a should have a sendable type in Γ_1, $\Gamma_1(b) \leq (\mathbf{s}(\sigma'))^{\mathtt{O}}$ for some σ'. Now suppose $\Gamma_1 \asymp_1 \Gamma_2$. Then by definition, $\Gamma_1 \sqcap \Gamma_2$ is defined, which is impossible. In this (hypothetical) environment the output capability associated with b must be dominated by the input capability, which would in turn require $\mathbf{s}(\sigma') \leq (\sigma)^{\mathtt{I}}$. However this constraint can never be satisfied for channel types.

EXAMPLE 4.9. If $\Gamma \vdash_1 \lambda().\,\mathbf{sq}(a):\tau$ then $\tau \notin \mathsf{Sble}$.

Because of the subsumption rule SUB the proof of this statement is not immediate. However first consider any derivation of $\Gamma' \vdash_1 \mathbf{sq}(a):\rho$. This will consist of a sequence of applications of SUB and an application of IN. By induction on this sequence we can establish that $\rho \notin \mathsf{Sble}$ and $\Gamma' \not\vdash_1 \mathtt{SBL}$; the case of an application of SUB is handled by the fact that Sble is downwards closed with respect to subtyping. In other words if we can derive $\Gamma' \vdash_1 \mathbf{sq}(a):\rho$ then ρ must be $\mathtt{proc}$.

Now let us examine a derivation of $\Gamma \vdash_1 \lambda().\,\mathbf{sq}(a):\tau$. This again will use the rules SUB, and ABS. We may also use TERM$_l$. By induction on the derivation we can show that $\tau \notin \mathsf{Sble}$. Because of our previous reasoning, τ must take the form either $\mathbf{s}(\tau' \to \mathtt{proc})$ by TERM$_l$ rule or $\tau' \to \mathtt{proc}$. The former is impossible since there is no $\Gamma \leq \Gamma'$ such that $\Gamma' \vdash_1 \mathtt{SBL}$ and $\Gamma', x:\mathtt{unit} \vdash_1 \mathbf{sq}(a)$ by the same argument in the above. The latter can not be in Sble because $\mathtt{proc}$ is not sendable; subsequent applications of SUB can not infer a sendable type, again because Sble is downwards closed with respect to subtyping.

EXAMPLE 4.10. Recall the first system in Example 3.5:

$$a?(x).\,P \mid b!\langle V\rangle \parallel b?(Y).\,Y\,()$$

where V is the value $\lambda().\,\mathbf{sq}(a)$.

Since V can not be assigned a sendable type, if the right-hand side location is typable, then b should be local by OUT$_l$. Then it can not be system-composable with the left-hand side where b is local too.

EXAMPLE 4.11. Recall the third system in Example 3.6.

$$d?(X).\,X() \parallel b!\langle\lambda().\,\mathbf{sq}(c)\rangle \mid b?(Y).\,d!\langle Y\rangle \mid c?(x).\,Q$$

Since d is not local in the left-hand side, if $b?(Y:\tau).\,d!\langle Y\rangle$ is typable for some τ then τ must be in Sble. Suppose Γ types this system and that $\Gamma(b)$ has the form $\langle(\tau_{\mathtt{I}}),(\tau_{\mathtt{O}})\rangle$. To be a valid type we need $\tau_{\mathtt{O}} \leq \tau_{\mathtt{I}}$ and since $\tau_{\mathtt{I}} \leq \tau \in \mathsf{Sble}$ we have $\tau_{\mathtt{O}} \in \mathsf{Sble}$. However the type assigned to $b!\langle\lambda().\,\mathbf{sq}(c)\rangle$, say τ', must satisfy $\tau' \leq \tau_{\mathtt{O}}$, that is τ' must be an element of Sble. From Example 4.9 we know this is impossible.

Note that this example demonstrates the need for $\mathbf{s}(\ )$ as a constructor on types. By asserting that the type on the bound variable Y is in Sble we can reject inappropriate inputs on the channel b. In a typing system based on the simple assertions of the form

$$\vdash \quad \tau \text{ is sendable}$$

where τ is a type from Section 2, such reasoning would be difficult.

SUBJECT REDUCTION In this subsection we prove locality is preserved under reduction. We take for granted that the revised type system satisfies the usual properties such as Narrowing, Weakening etc.; these may be checked by the reader.

LEMMA 4.12.

(1) (Algebra on environments) $\Gamma_1 \vdash \mathtt{SBL}$ *and* $\Gamma_2 \vdash \mathtt{SBL}$ *imply* $\Gamma_1 \sqcap \Gamma_2 \vdash \mathtt{SBL}$, and $\Delta_1 \asymp_1 \Delta_2$ and $\Delta_1 \asymp_1 \Delta_3$ implies $\Delta_1 \asymp_1 \Delta_2 \sqcap \Delta_3$.
(2) (Sendable types) If $\rho \in \mathsf{Sble}$ then $\Gamma \vdash_1 P : \rho$ *implies there exists* Δ *s.t.* $\Delta \geq \Gamma$, $\Delta \vdash_1 \mathtt{SBL}$, *and* $\Delta \vdash_1 P : \rho$.
(3) (Local substitution) *Suppose* $\Gamma, x : \tau \vdash_1 P : \rho$ *and* $\Gamma \vdash_1 V : \tau$. *Then we have* $\Gamma \vdash_1 P\{V/x\} : \rho$.

Proof See Appendix B. □

The proof of the second property relies directly on the constraint on the construction of arrow types: $\tau \to \rho$ with $\rho \in \mathsf{Sble}$ is only a valid type if $\tau \in \mathsf{Sble}$. Relaxing this constraint would allow us to type many more terms as sendable. Typical examples take the form $(\lambda xy.x)PQ$, with P being sendable and Q non-sendable. Indeed such terms may be exported between locations without violating locality constraints. But inventing a typing system which allows this behaviour is a topic for further serious research (cf. [26]).

The second property is also the most important to prove the third. In the type system there are many different ways of inferring a sendable type, for example using CONST$_l$, CHAN$_l$, SUB or APP. However we can regard all sendable types as being inferred in a uniform manner by an application of TERM$_l$.

Note that this lemma does *not* alleviate the need for the constructor $\mathbf{s}(\ )$ on types. We have already seen in Example 4.11 how it is used to eliminate potential mistyping. It is also used in the following more complicated example which shows that the first system in Example 3.6 is untypable.

EXAMPLE 4.13. Recall the first system in Example 3.6.

$$d?(X).\, X() \,|\, b!\langle V\rangle \;\|\; b?(Y).\, d!\langle\lambda().(Y\,c)\rangle \,|\, c?(x).\, Q$$

with V is the sendable value $\lambda(x).\,\mathbf{sq}(x)$. Suppose this is typed as $\lambda(x{:}\sigma_0).\,\mathbf{sq}(x)$ with some σ_0. Then by definition of $\mathbf{sq}(x)$ in Example 2.4, σ_0 should satisfy $\sigma_0 \leq (\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{I}}$.

Now suppose Γ types the above system, and $\Gamma(b)$ has the form $\langle(\tau_{\mathtt{I}}), (\tau_{\mathtt{O}})\rangle$. Assume $b?(Y{:}\tau').\, d!\langle\lambda().(Y\,c)\rangle$ is typable for some τ'. Since V should be sendable in the right hand side location, to infer $b!\langle V\rangle$, there should be $\Gamma(b) \leq (\mathbf{s}(\sigma_0' \to \mathtt{proc}))^{\mathtt{O}}$ for some $\sigma_0' \leq \sigma_0$. Hence we require $\mathbf{s}(\sigma_0' \to \mathtt{proc}) \leq \tau_{\mathtt{O}} \leq \tau_{\mathtt{I}} \leq \tau'$.

Next, since d is not local in the right-hand side, by Lemma 4.12 (2), there exists Δ such that $\Delta \vdash Y\,c : \mathtt{proc}$ with $\Delta \vdash \mathtt{SBL}$. We show there does not exist such Δ. Suppose $\Delta \vdash Y\,c : \mathtt{proc}$. Then we must have the following derivation.

$$\dfrac{\dfrac{\Delta \vdash_1 Y : \tau' \quad \tau' \leq \sigma' \to \mathtt{proc}}{\Delta \vdash_1 Y : \sigma' \to \mathtt{proc} \quad \Delta \vdash_1 c : \sigma'}}{\Delta \vdash_1 Y\,c : \mathtt{proc}}$$

By $\mathtt{proc} \notin \mathsf{Sble}$, $\mathbf{s}(\sigma_0' \to \mathtt{proc}) \leq \sigma' \to \mathtt{proc}$ can be derived only by $\sigma_0' \to \mathtt{proc} \leq \sigma' \to \mathtt{proc}$ by the definition of $\leq$. Thus we should have: $\sigma' \leq \sigma_0' \leq \sigma_0 \leq (\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{I}}$ by the ordering of arrow types. Since $\Delta(c) \leq \sigma'$, $\Delta(c)$ should have the input capability, which means Δ in the above should be non-sendable. Thus this system is not typable under any environment Γ. □

The main lemma requires the order-theoretic property, FBC, of our subtyping relation, together with Lemma 4.12.

LEMMA 4.14. **(Main Lemma)** *Suppose* $\Gamma_1, x : \tau' \vdash_1 P : \pi$ *and* $\Gamma_2 \vdash_1 V : \mathbf{s}(\tau)$ *with* $\Gamma_1 \asymp_1 \Gamma_2$ *and* $\tau' \geq \mathbf{s}(\tau)$. *Then there exists* Δ *such that:*

(1) $\Gamma_2 \leq \Delta$ with $\Delta \vdash_1 \mathtt{SBL}$ and $\Delta \vdash_1 V : \mathbf{s}(\tau)$,
(2) $\Gamma_1 \sqcap \Delta \vdash_1 P\{V/x\} : \pi$, and
(3) $\Gamma_1 \sqcap \Delta \sqcap \Gamma_2 = \Gamma_1 \sqcap \Gamma_2$ with $\Gamma_1 \sqcap \Delta \asymp_1 \Gamma_2$.

PROOF. **(1)** By Lemma 4.12 (2).

(2) Let us define $\Gamma = \Gamma_1 \asymp_1 \Gamma_2$. Then we have $\Gamma \leq \Gamma_1$ and $\Gamma \leq \Gamma_2 \leq \Delta$, which implies that $\Gamma_1 \sqcap \Delta$ defined and $\Gamma_1 \sqcap \Delta \leq \Gamma_1$. Then by the standard narrowing property, $\Gamma_1 \sqcap \Delta, x{:}\tau' \vdash P : \rho$ and $\Gamma_1 \sqcap \Delta \vdash V : \mathbf{s}(\tau)$. Hence $\Gamma_1 \sqcap \Delta \vdash V : \tau'$ by SUB. Applying Lemma 4.12 (3) to these, we have done.

(3) First by the proof of (2) above, we know $\Gamma_1 \asymp \Delta$. Then by the commutativity of $\sqcap$, we have $(\Gamma_1 \sqcap \Delta) \sqcap \Gamma_2 = \Gamma_1 \sqcap (\Delta \sqcap \Gamma_2) = \Gamma_1 \sqcap \Gamma_2$. Since $\Delta \vdash_1 \mathtt{SBL}$, $\Delta \asymp_1 \Gamma_{1,2}$. Then by Lemma 4.12 (1), we have $\Gamma_1 \sqcap \Delta \asymp_1 \Gamma_2$, as required. □

LEMMA 4.15. (Struct)

- *For processes* P, Q *if* $\Gamma \vdash_1 P : \pi$ *and* $P \equiv Q$, *then* $\Gamma \vdash_1 Q : \pi$.
- *For systems* N, M *if* $\Gamma \vdash_1 N$ *and* $N \equiv M$, *then* $\Gamma \vdash_1 M$.

Proof The proof of both statements are virtually identical. We examine the only non-trivial case (for systems), the scope opening rule.

Suppose $\Gamma \vdash_1 (\nu a{:}\sigma)M \parallel N$. Then by induction, there are Γ_1 and Γ_2 which satisfy $\Gamma_1, a:\sigma \vdash_1 M$, $\Gamma_2 \vdash_1 N$ and $\Gamma_1 \asymp_1 \Gamma_2$. Since $a \notin \mathsf{dom}(\Gamma_2)$, we have $\Gamma_1, a:\sigma \asymp_1 \Gamma_2$. Hence we have $\Gamma_1, a:\sigma \sqcap \Gamma_2 \vdash_1 M \parallel N$ which implies, since $\Gamma_1, a:\sigma \sqcap \Gamma_2 = \Gamma_1 \sqcap \Gamma_2, a:\sigma$, that $\Gamma_1 \sqcap \Gamma_2 \vdash_1 (\nu a{:}\sigma)(M \parallel N)$, as required.

For the other direction, suppose $\Gamma \vdash_1 (\nu a{:}\sigma)(M \parallel N)$ with $a \notin \mathsf{fn}(N)$. Then by induction, we have: $\Gamma'_1 \vdash_1 M$ and $\Gamma'_2 \vdash_1 N$ such that $\Gamma'_1 \sqcap \Gamma'_2 = \Gamma, a:\sigma$. Since $a \notin \mathsf{fn}(N)$, we know that $\Gamma'_2/a \vdash_1 N$. Then by $(\Gamma'_1 \sqcap \{a:\sigma\})(a) = \sigma$, we have $\Gamma'_1 \sqcap \{a:\sigma\} \vdash_1 M$, which implies $\Gamma'_1/a \vdash_1 (\nu a{:}\sigma)M$. Finally by $\Gamma'_1/a \asymp_1 \Gamma'_2/a$, we have done. $\square$ $\square$

THEOREM 4.16. **(Subject Reduction Theorem)**

- *If* $\Gamma \vdash_1 P:\rho$ *and* $P \longmapsto Q$, *then* $\Gamma \vdash_1 Q:\rho$.
- *If* $\Gamma \vdash_1 N$ *and* $N \longrightarrow M$, *then* $\Gamma \vdash_1 M$.

Proof The proof for processes is identical to that of Theorem 2.6 and therefore we concentrate on that for systems. We use induction on the derivation of $N \longrightarrow M$ and because of the previous Lemma, and the first part of the Theorem, there are only two non-trivial cases:

(1) $(\cdots Q \mid \mathtt{Spawn}(P)) \longrightarrow (\cdots Q) \parallel P$

(2) $(u?(\tilde{x}{:}\tilde{\tau}).P \mid \cdots) \parallel (u!\langle\tilde{V}\rangle Q \mid \cdots) \longrightarrow (P\{\tilde{V}/\tilde{x}\} \mid \cdots) \parallel (Q \mid \cdots)$.

Case (1): Assume $\Gamma \vdash_1 (\cdots Q \mid \mathtt{Spawn}(P))$ and therefore in particular that $\Gamma \vdash_1 \mathtt{Spawn}(P) : \mathtt{proc}$. Then $\Gamma \vdash_1 P : \mathbf{s}(\mathtt{proc})$. Hence by Lemma 4.12 (2), there exists $\Delta \vdash_1 \mathtt{SBL}$ such that $\Gamma \le \Delta$ and $\Delta \vdash_1 P : \mathtt{proc}$. It follows that $\Gamma \vdash_1 (\cdots Q) \parallel P$ since $\Gamma = \Gamma \sqcap \Delta$ and $\Gamma \asymp_1 \Delta$.

Case (2): Let us just consider the case

$$u?(\tilde{x}{:}\tilde{\tau}).\,P \parallel u!\langle\tilde{V}\rangle.\,Q \longrightarrow P\{\tilde{V}/\tilde{x}\} \parallel Q \tag{4.1}$$

We know that Γ has the form $\Gamma_1 \sqcap \Gamma_2$ where $\Gamma_1 \vdash_1 u?(\tilde{x}{:}\tilde{\tau}).\,P : \mathtt{proc}$ and $\Gamma_2 \vdash_1 u!\langle\tilde{V}\rangle.\,Q : \mathtt{proc}$ with $\Gamma_1 \asymp_1 \Gamma_2$. We show that for some Γ'_1 such that $\Gamma_1 \sqcap \Gamma_2 = \Gamma'_1 \sqcap \Gamma_2$ and $\Gamma'_1 \asymp_1 \Gamma_2$, we have $\Gamma'_1 \vdash_1 P\{\tilde{V}/\tilde{x}\}$ and $\Gamma_2 \vdash_1 Q$.

First, since u is not local in the left-hand side location, $\Gamma_2 \vdash_1 u!\langle\tilde{V}\rangle.\,Q : \mathtt{proc}$ should be inferred by OUT_d. Hence we have the following derivations, for each V_i in $\tilde{V}$ $(0 \le n)$:

$$\Gamma_2 \vdash u : (\mathbf{s}(\tilde{\tau}'))^{\mathtt{O}}, \quad \Gamma_2 \vdash u : V_i : \mathbf{s}(\tau'_i) \quad \text{and} \quad \Gamma_2 \vdash Q : \mathtt{proc} \tag{4.2}$$

By Lemma 4.12 (2), there exists Δ_i such that $\Delta_i \vdash_1 \mathtt{SBL}$ and $\Delta_i \vdash_1 V_i : \mathbf{s}(\tau'_i)$. Set $\Delta = \sqcap \Delta_i$. Then we have $\Delta \ge \Gamma_1$ and $\Delta \vdash_1 \mathtt{SBL}$ by Lemma 4.12 (1). The required Γ'_1 is $\Gamma_1 \sqcap \Delta$, and by Lemma 4.14 (3), we have: $\Gamma_1 \sqcap \Gamma_2 = \Gamma'_1 \sqcap \Gamma_2$ with $\Gamma'_1 \asymp_1 \Gamma_2$, but we need to prove $\Gamma'_1 \vdash_1 P\{\tilde{V}/\tilde{x}\}$.

From $\Gamma_1 \vdash_1 u?(\tilde{x}{:}\tilde{\tau}).\,P : \mathtt{proc}$ we have

$$\Gamma_1 \vdash u : (\tilde{\tau})^{\mathtt{I}} \quad \text{and} \quad \Gamma_1,\ \tilde{x}{:}\tilde{\tau} \vdash P : \mathtt{proc} \tag{4.3}$$

Now assume $(\Gamma_i \sqcap \Gamma_j)(u) = \langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$ (Note we do not have $\mathbf{s}(\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle)$ since $S_{\mathtt{I}} \ne \top$). Then by $\Gamma_1 \sqcap \Gamma_2 \le \Gamma_1$ and $\Gamma_1 \sqcap \Gamma_2 \le \Gamma_2$, we know $\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle \le \Gamma_1(u) \le (\tilde{\tau})^{\mathtt{I}}$ and $\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle \le \Gamma_2(u) \le (\mathbf{s}(\tilde{\tau}'))^{\mathtt{O}}$. Since $S_{\mathtt{I}} \ge S_{\mathtt{O}}$ by definition, we get $(\tilde{\tau}) \ge (\mathbf{s}(\tilde{\tau}'))$, hence $\tau_i \ge \mathbf{s}(\tau'_i)$ for each i. Therefore we apply Lemma 4.14 (2) for each i to obtain the required $\Gamma'_1 \vdash_1 P\{\tilde{V}/\tilde{x}\} : \mathtt{proc}$. $\square$

COROLLARY 4.17. (Type Safety) $\Gamma \vdash_1 N$ *and* $N \longrightarrow\!\!\!\!\!\rightarrow M$ *imply* $M \stackrel{lerr}{\not\longrightarrow}$.

Proof Theorem 3.3 can be trivially adapted to the judgements $\Gamma \vdash_1 N$. $\square$

5 Further Development

Here we examine some issues which arise naturally from the ideas introduced in the previous section.

5.1 Generalisation to Global/Local Subtyping

Our typing system has a static view of the role of channels. From the point of view of a given location they can only be used for input *locally* whereas there is *global* access to its output capability. A more general view is proposed in [34], whereby the input/output capabilities of each channel can be designated to be either global or be restricted to being local. Here we show that our typing system can be adapted to this more general framework.

In this extension channel types are labelled by one of the *locality modes*, $\{\mathtt{GG}, \mathtt{LG}, \mathtt{GL}, \mathtt{LL}\}$, ranged over by $m, m', \ldots$. Their meaning is as follows:

- $\mathtt{GG}$ – a channel is allowed to be used as the input and output subjects anywhere.
- $\mathtt{GL}$ (resp. $\mathtt{LG}$) – a channel is used as the input (resp. output) subject anywhere, while as the output (resp. input) subject only inside this location.
- $\mathtt{LL}$ – a channel is used as the input and output subjects only in this location.

A partial order on these modes is given by:

$$\begin{tikzcd} & \mathtt{LL} & \\ \mathtt{GL} \arrow[ur] & & \mathtt{LG} \arrow[ul] \\ & \mathtt{GG} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

and this in turn is reflected in the ordering on the extended channel types, which are given by $m\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$. Here m denotes how the channel is used as the subject while $S_{\mathtt{I}}, S_{\mathtt{O}}$ in $\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$ stand for the types of *objects* which it carries.

We now briefly outline how our typing system can be adapted to these more general types; In the revised system judgements take the form

$$\Gamma \vdash_{\mathrm{g}} P : \rho$$

First we need a more general replacement for the rule $\textsc{Chan}_l$ in Figure 9 to indicate when channels, or more precisely channel capabilities, may themselves be transferred between locations.:

$$\textsc{Chan}_g \quad \begin{array}{cc} \dfrac{\Gamma \vdash_{\mathrm{g}} a : \mathtt{LL}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle}{\Gamma \vdash_{\mathrm{g}} a : \mathbf{s}(\mathtt{GG}\langle \top, \bot\rangle)} & \dfrac{\Gamma \vdash_{\mathrm{g}} a : \mathtt{LG}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle}{\Gamma \vdash_{\mathrm{g}} a : \mathbf{s}(\mathtt{GG}\langle \top, S_{\mathtt{O}}\rangle)} \\[2ex] \dfrac{\Gamma \vdash_{\mathrm{g}} a : \mathtt{GL}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle}{\Gamma \vdash_{\mathrm{g}} a : \mathbf{s}(\mathtt{GG}\langle S_{\mathtt{I}}, \bot\rangle)} & \dfrac{\Gamma \vdash_{\mathrm{g}} a : \mathtt{GG}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle}{\Gamma \vdash_{\mathrm{g}} a : \mathbf{s}(\mathtt{GG}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle)} \end{array}$$

In general a capability can only be transmitted if it has the form $\mathtt{GG}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$ for some $S_{\mathtt{I}}, S_{\mathtt{O}}$; moreover the input type $S_{\mathtt{I}}$ and the output type $S_{\mathtt{O}}$ is determined by the mode of the channel type. As an example if this is $\mathtt{GL}$, then it is prohibited from being used as the output subject in an other location; hence it can only be sent as the capability $\langle S_{\mathtt{I}}, \bot\rangle$, with the mode $\mathtt{GG}$.

The input/output rules require minor modifications:

$$\textsc{In}_{gl}: \dfrac{\begin{array}{c}\Gamma \vdash_{\mathrm{g}} u : \mathtt{LL}(\tilde{\tau})^{\mathtt{I}} \\ \Gamma, \tilde{x}:\tilde{\tau} \vdash_{\mathrm{g}} P : \mathtt{proc}\end{array}}{\Gamma \vdash_{\mathrm{g}} u?(\tilde{x}:\tilde{\tau}).P : \mathtt{proc}}$$

$$\textsc{Out}_{gl}: \dfrac{\begin{array}{l}\Gamma \vdash_{\mathrm{g}} u : \mathtt{LL}\langle(\tilde{\tau}'), (\tilde{\tau})\rangle \\ \Gamma \vdash_{\mathrm{g}} V_i : \tau_i \\ \Gamma \vdash_{\mathrm{g}} P : \mathtt{proc}\end{array}}{\Gamma \vdash_{\mathrm{g}} u!\langle\tilde{V}\rangle P : \mathtt{proc}} \qquad \textsc{Out}_{gd}: \dfrac{\begin{array}{l}\Gamma \vdash_{\mathrm{g}} u : \mathtt{LL}(\mathbf{s}(\tilde{\tau}))^{\mathtt{O}} \\ \Gamma \vdash_{\mathrm{g}} V_i : \mathbf{s}(\tau_i) \\ \Gamma \vdash_{\mathrm{g}} P : \mathtt{proc}\end{array}}{\Gamma \vdash_{\mathrm{g}} u!\langle\tilde{V}\rangle P : \mathtt{proc}}$$

The remaining rules of the local typing system remain unchanged. However in this extended type system we rely entirely on the rule $\textsc{Term}_l$ to infer processes have the sendable type $\mathbf{s}(\mathtt{proc})$. This may be alleviated somewhat by the two optional rules:

$$\dfrac{\begin{array}{c}\Gamma \vdash_{\mathrm{g}} u : \mathtt{GL}(\tilde{\tau})^{\mathtt{I}} \\ \Gamma, \tilde{x}:\tilde{\tau} \vdash_{\mathrm{g}} P : \mathbf{s}(\mathtt{proc})\end{array}}{\Gamma \vdash_{\mathrm{g}} u?(\tilde{x}:\tilde{\tau}).P : \mathbf{s}(\mathtt{proc})} \qquad \dfrac{\begin{array}{l}\Gamma \vdash_{\mathrm{g}} u : \mathtt{LG}(\mathbf{s}(\tilde{\tau}))^{\mathtt{O}} \\ \Gamma \vdash_{\mathrm{g}} V_i : \mathbf{s}(\tau_i) \quad \Gamma \vdash_{\mathrm{g}} P : \mathbf{s}(\mathtt{proc})\end{array}}{\Gamma \vdash_{\mathrm{g}} u!\langle\tilde{V}\rangle P : \mathbf{s}(\mathtt{proc})}$$

Finally to extend a system to a composition of systems we need a more general definition of *system composable*.

DEFINITION 5.1. Then two environments Γ_1 and Γ_2 are composable, denoted by $\Gamma_1 \asymp_{\mathrm{g}} \Gamma_2$, if $\Gamma_1 \asymp \Gamma_2$ and if $u : m_i\langle S_{i\mathtt{I}}, S_{i\mathtt{O}}\rangle \in \Gamma_i$ $(i = 1, 2)$, then (1) $m_i = \mathtt{LL}$ implies $S_{j\mathtt{I}} = \top$ and $S_{j\mathtt{O}} = \bot$, (2) $m_i = \mathtt{LG}$ implies $S_{j\mathtt{I}} = \top$, and (3) $m_i = \mathtt{GL}$ implies $S_{j\mathtt{O}} = \bot$ with $i \neq j$. □

This leads to the final change to the typing rules:

$$\textsc{Par}_g: \quad \dfrac{\Gamma \vdash_{\mathrm{g}} M \quad \Delta \vdash_{\mathrm{g}} N \quad \Gamma \asymp_{\mathrm{g}} \Delta}{\Gamma \sqcap \Delta \vdash_{\mathrm{g}} M \,\|\, N}$$

THEOREM 5.2. (Subject Reduction) *If* $\Gamma \vdash_{\mathrm{g}} N$ *and* $N \longrightarrow\!\!\!\!\rightarrow M$, *then* $\Gamma \vdash_{\mathrm{g}} M$.

Proof Omitted. □

The definition of run-time error is however somewhat more complicated. In general whether or nor there is a violation of the locality requirements depends on an apriori decision of which channel capabilities can be used globally. For example if a typing dictates that input on a channel a is global then no run-time error occurs if the input prefix $a?x$ occurs at two distinct locations.

Thus to formalise run-time errors we require a notion of a *tagged* version of the language along the lines of [27] or [15]. The reader familiar with this technique should be easily convinced that such a tagged language could be developed, together with an appropriate version of a Type Safety Theorem.

5.2 Behavioral Equivalence

Typing systems impose constraints on the communication structure of processes and various authors, for example [27, 32, 37] have used this to define relativised behavioural equivalences. These have proved useful for example in studying the properties of translations between languages, [22, 3, 32, 37].

This technique can also be applied to $\mathrm{D}\pi\lambda$, thereby opening up the possibility of obtaining interesting relativised behavioural theories for higher-order processes. Let $\approx_\Gamma$ (resp. $\sim_\Gamma$) denote a typed weak (resp. strong) barbed reduction-closed congruence defined by input/output predicates and reduction-closure property as in [27, 37, 2, 32, 18]. Various properties of $\approx_\Gamma$ and $\sim_\Gamma$, proved for variations of the π-calculus, can easily be generalised to $\mathrm{D}\pi\lambda$; a simple example is closure under β-reduction (see page 10 in [2]). We can also prove various distributed equations, such as

$$\begin{array}{c}(P \,|\, \mathtt{Spawn}(Q)) \,\|\, R \approx_\Delta (P \,\|\, Q) \,\|\, R \sim_\Delta P \,\|\, (Q \,|\, R) \\ P \,\|\, (a!\langle\tilde{V}\rangle \,|\, Q) \sim_\Delta (P \,|\, a!\langle\tilde{V}\rangle) \,\|\, Q\end{array}$$

We also have the following multiple higher-order strong replication theorem:

PROPOSITION 5.3. *Let us define* $R \stackrel{\mathrm{def}}{=} *a?(\tilde{x}).\, R_1 \,|\, \cdots \,|\, *a?(\tilde{x}).\, R_n$ *with* R_i *sendable. Then we have:* $(\nu a)(R \,\|\, P \,\|\, Q) \;\sim_\Gamma\; (\nu a)(R \,|\, P) \,\|\, (\nu a)(R \,|\, Q)$

Proof As in [27, 37], we only have to consider a closure of the parallel composition with a system (note that systems are closed under only $\|$ and ν). We take the set $\mathbf{R}$ of all pairs of the form

$$\langle (N \,\|\, (\nu a)(R \,\|\, P \,\|\, Q)), (N \,\|\, (\nu a)(R \,|\, P) \,\|\, (\nu a)(R \,|\, Q)) \rangle \tag{5.4}$$

where both systems in the pair are typable under environment Γ. We show this relation is a strong barbed reduction-closed up to $\sim_\Gamma$ and restriction.

To establish this, we first define the same notion of Definition 5.3.1 in [27]; we say *a is only used as a trigger in P* if $\Gamma \vdash C[P] : \mathtt{proc}$ and there exists Δ s.t. $\Delta \geq \Gamma$, $\Delta \vdash_1 P : \mathtt{proc}$ and $\Delta(a) = \langle \top, S_0 \rangle$ or $\Delta(a) \leq \mathbf{s}(\tau)$. Then we observe that:

1. Suppose R is sendable. Then $\Gamma \vdash_1 (P | R) \parallel Q$ iff $\Gamma \vdash_1 P \parallel R \parallel Q$, and $(P | R) \parallel Q \sim_\Gamma P \parallel R \parallel Q$.
2. If R is sendable, then all free names in R are only used as a trigger.
3. a is only used as a trigger in P and Q since the left-hand side of the above equation is typable. Hence P and Q may only export the sendable value V via a.
4. If a is used as a trigger in Q and V is sendable, then a in $Q\{V/X\}$ is again only used as a trigger.

Now take the set $\mathbf{R}'$ of typable pairs of the form

$$\langle (\nu a)(N\{a/a'\} \parallel R \parallel P\{a/a'\} \parallel Q\{a/a'\}), (\nu a, a')(N \parallel (R | P) \parallel (R\{a'/a\} | Q)) \rangle$$

where $a' \notin \mathsf{fn}(R)$ and a and a' are used as triggers in N,R_i, P and Q (note this is a simple extension of the equation (25) in Appendix B in [27]). To establish $\mathbf{R}$ is in $\sim_\Gamma$, we show the above relation $\mathbf{R}'$ is again a strong barbed reduction-closed up to $\sim_\Gamma$, using 1 to 4 above. □

Note we do not require any side condition for P and Q, for example stating that P and Q must be of a certain syntactic form; instead the typing system enforces implicit constraints on the various components of the systems. Note also that this proposition can not be derived in the framework of [32] since a is neither a linear nor an ω-receptive name.

Such theorems will be useful for reasoning about object-oriented systems where templates are shared among locations. Further extension of typed equivalences studied in π-calculus (e.g. [32, 37]) to distributed higher-order processes is an interesting research topic which we intend to pursue.

5.3 Type Checking

For a practical use of a typing system, it is essential that we can check the well-typedness of a system N against a global type environment Γ. For this purpose, first we propose a typing system ($\mathbf{Min}_{\pi\lambda}$) which can induce the minimum type of a given term P and Γ (if it has a type) in $\pi\lambda$ in Section 2, by deleting SUB and modifying APP, OUT and IN in Figure 4 as follows.

$$\textsc{App}_m: \quad \frac{\Gamma \vdash P : \tau_1 \to \rho \quad \Gamma \vdash Q : \tau_2 \quad \tau_2 \leq \tau_1}{\Gamma \vdash PQ : \rho}$$

$$\textsc{In}_m: \quad \frac{\Gamma(u) \leq (\tilde{\tau})^{\mathtt{I}} \quad \Gamma, \tilde{x}:\tilde{\tau} \vdash P : \mathtt{proc}}{\Gamma \vdash u?(\tilde{x}:\tilde{\tau}).P : \mathtt{proc}} \qquad \textsc{Out}_m: \quad \frac{\Gamma \vdash V_i : \tau_i \quad \Gamma(u) \leq (\tilde{\tau})^{\mathtt{O}} \quad \Gamma \vdash P : \mathtt{proc}}{\Gamma \vdash u!\langle \tilde{V} \rangle P : \mathtt{proc}}$$

PROPOSITION 5.4.

(1) (ordering) $\rho_1 \leq \rho_2$ is decidable.
(2) (the minimum type in $\pi\lambda$) In the typing system in Figure 4, if $\Gamma \vdash P : \rho$ then there exists ρ' such that $\Gamma \vdash P : \rho'$ and, for any ρ_0, if $\Gamma \vdash P : \rho_0$, then $\rho' \leq \rho_0$.
(3) (algorithm) There is a type-checking algorithm that given type environment Γ and term P, computes the minimum type ρ such that $\Gamma \vdash P : \rho$ if one exists.

PROOF. **(1)** we first note functional types of our system correspond the regular system in [20] (see Theorem 5 in [20]). Next we observe that a subtyping between channel types $\langle S_{1\mathtt{I}}, S_{1\mathtt{O}} \rangle \leq \langle S_{2\mathtt{I}}, S_{2\mathtt{O}} \rangle$ are also computed in the same way as the arrow types; hence the subtyping relation is decidable.

(2) by easy induction on the derivations of $\Gamma \vdash P : \rho$ in $\mathbf{Min}_{\pi\lambda}$, we obtain (a) if $\Gamma \vdash P : \rho$ is derived from $\mathbf{Min}_{\pi\lambda}$, then it is also derived from the system in Figure 4, (b, unique type) $\Gamma \vdash P : \rho_1$ and $\Gamma \vdash P : \rho_2$ are derived in $\mathbf{Min}_{\pi\lambda}$, then $\rho_1 = \rho_2$, and (c, $\mathbf{Min}_{\pi\lambda}$ has smaller types) If $\Gamma \vdash P : \rho$ is derived from the system in Figure 4, then $\Gamma \vdash P : \rho'$ is derived from $\mathbf{Min}_{\pi\lambda}$ for some ρ' such that $\rho' \leq \rho$. By (a,b,c), (2) is straightforward.

(3) It can be constructed straightforwardly based on $\mathbf{Min}_{\pi\lambda}$. See Appendix C for that algorithm. □

To construct the typing checking algorithm for the local D$\pi\lambda$ in Section 4, we need to eliminate not only the subsumption rule, but also $\textsc{Term}_l$ in Figure 9 to obtain a syntax directed system. The basic idea is to keep the sendable flag $\mathbf{s}(\ )$ as much as possible during inference using the partial meet operator. We define a typing system, $\mathbf{Sbl}_{\mathrm{D}\pi\lambda}$, in Figure 10. In $\mathbf{Sbl}_{\mathrm{D}\pi\lambda}$, we assume every type takes either of the following three forms, which correspond (1,2,3) in Lemma 4.4, respectively ($k \geq 0$).

(1) $\mathbf{s}(\mathbf{s}(\tau_1) \to \mathbf{s}(\mathbf{s}(\tau_2) \to \mathbf{s}(\mathbf{s}(\tau_3) \to \cdots \to \mathbf{s}(\rho_1)) \cdots))$,
(2) $\tau_1 \to \tau_2 \to \tau_3 \to \cdots \to \rho_G$, or
(3) $\mathbf{s}(\tau_1 \to \tau_2 \to \tau_3 \to \cdots \to \rho_G)$

where ρ_1 takes the form of (2) and ρ_G is either $\mathtt{proc}$, $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$ or a base type. We use the following function, $\mathtt{up}(\rho_0)$ which changes ρ_0 to the least non-sendable type ρ' such that $\rho_0 \leq \rho'$.

$$\begin{array}{lcll} \mathtt{up}(\rho_0) & = & \mathbf{s}(\tau_1) \to \mathbf{s}(\tau_2) \cdots \mathbf{s}(\tau_n) \to \rho_1 & \text{if } \rho_0 \text{ is in the form of (1) above} \\ \mathtt{up}(\rho_0) & = & \rho & \text{else if } \rho_0 = \mathbf{s}(\rho) \\ \mathtt{up}(\rho_0) & = & \rho_0 & \text{else} \end{array}$$

We can easily observe that if $\rho \lneq \rho'$, then $\mathtt{up}(\rho) \notin \mathsf{Sble}$ and $\rho \leq \mathtt{up}(\rho) \leq \rho'$.

Then we have the following propositions.

PROPOSITION 5.5.

(1) $\Gamma_1 \asymp \Gamma_2, \Gamma_1 \asymp_1 \Gamma_2$ and $\Gamma \vdash_1 \mathtt{SBL}$ are decidable.
(2) (sendable) Given Γ and P, there is an algorithm that computes $\Gamma \vdash_1 P : \mathbf{s}(\rho)$ for some ρ if one exists.
(3) (algorithm) Given Γ and N, there is a type-checking algorithm that computes $\Gamma \vdash_1 N$ if one exists.

PROOF. (1) First, given ρ_1 and ρ_2, it is decidable to check whether $\rho_1 \sqcap \rho_2$ is defined or not since $\sqcap$ and $\sqcup$ are directly defined in § 2.3 and § 4.1, and the size of substructures of types in these definitions is always decreased. Then $\Gamma_1 \asymp \Gamma_2$ is decidable since this problem is reducible to check definedness of $\Gamma_1(a) \sqcap \Gamma_2(a)$ for each $a \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$. The second case is easy from the first. The third case is obvious. For (2), we note that (a) $\Gamma \Vdash_1 P : \mathbf{s}(\rho)$ implies $\Gamma \vdash_1 \mathtt{SBL}$ and (b) if $\Gamma \Vdash_1 P : \rho$, then $\Gamma \vdash_1 P : \rho$, and (c) If $\Gamma \vdash P : \rho$, then $\Gamma' \Vdash_1 P : \rho'$ is derived for some ρ', Γ' such that $\rho' \leq \rho$, and $\Gamma \leq \Gamma'$. Then (2) is constructed based on the rules in Figure 10. (3) is given using (2), together with the fact that $\Gamma_1 \asymp_1 \Gamma_2$ is decidable. See Appendix C for this algorithm. □

6 Discussion and Related Work

We have proposed a static local typing system for a simple higher-order distributed process language D$\pi\lambda$ and we used it to show that a global safety condition can be guaranteed by static type-checking of each local configuration. Our typing system does not require additional information on the resources available at different locations to ensure that higher-order processes can be safely passed between locations without violating locality constraints on channels. The notion of *sendable* values and the corresponding *sendable* types plays an essential role in our typing system. Other schemes for restricting capabilities of higher-order terms by types may also be found in various different contexts; for example, in reference types [26], agent migration [30], an implementation of network protocols [19], and a location-based Linda language [25].

The distributed component of D$\pi\lambda$ is rather primitive, but we believe that the type inference system can easily be adapted to languages where, for example, locations can be named and dynamically generated, such as those discussed in [3, 15], or where there is more significant interplay between the concurrent

Common Rules:

$$\textsc{Con}_s: \quad \Vdash_1 1 : \mathbf{s}(\mathtt{nat}) \quad \text{etc.} \qquad \textsc{Var}_s: \quad x:\tau \Vdash_1 x : \tau$$

$\textsc{Chan}_s$:

$$(1):\ a:\langle\top, S_{\mathtt{O}}\rangle \Vdash_1 a : \mathbf{s}(\langle\top, S_{\mathtt{O}}\rangle) \qquad (2):\ a:\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle \Vdash_1 a : \langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$$

$\textsc{Weak}_s$:

$$(1)\ \frac{\Gamma \Vdash_1 P : \mathbf{s}(\rho)}{u:\mathbf{s}(\tau), \Gamma \Vdash_1 P : \mathbf{s}(\rho)} \qquad (2)\ \frac{\Gamma \Vdash_1 P : \mathbf{s}(\rho)}{a:\langle\top, S\rangle, \Gamma \Vdash_1 P : \mathbf{s}(\rho)} \qquad (3)\ \frac{\Gamma \Vdash_1 P : \rho}{u:\tau, \Gamma \Vdash_1 P : \mathtt{up}(\rho)}$$

Functional Rules:

$\textsc{Abs}$:

$$(1)\ \frac{\Gamma, x:\mathbf{s}(\tau) \Vdash_1 P : \mathbf{s}(\rho)}{\Gamma \Vdash_1 \lambda(x:\tau).P : \mathbf{s}(\mathbf{s}(\tau) \to \mathbf{s}(\rho))} \qquad (2)\ \frac{\Gamma, x:\tau \Vdash_1 P : \rho \quad \Gamma \vdash_1 \mathtt{SBL}}{\Gamma \Vdash_1 \lambda(x:\tau).P : \mathbf{s}(\tau \to \rho)} \qquad (3)\ \frac{\Gamma, x:\tau \Vdash_1 P : \rho}{\Gamma \Vdash_1 \lambda(x:\tau).P : \tau \to \rho}$$

$$\textsc{App}_s: \quad \frac{\begin{array}{c}\tau = \tau_1 \to \rho \quad \text{or} \quad \tau = \mathbf{s}(\tau_1 \to \rho) \\ \Gamma_1 \Vdash_1 P : \tau \quad \Gamma_2 \Vdash_1 Q : \tau_2 \quad \tau_2 \leq \tau_1 \quad \Gamma_1 \asymp \Gamma_2\end{array}}{\Gamma_1 \sqcap \Gamma_2 \Vdash_1 PQ : \rho}$$

Process Rules:

$$\textsc{Nil}_s: \quad \Vdash_1 \mathbf{0} : \mathbf{s}(\mathtt{proc})$$

$\textsc{Out}_s$:

$$(1)\ \frac{\begin{array}{l}\Gamma(u) \leq (\mathbf{s}(\tilde{\tau}))^{\mathtt{O}} \\ \Gamma_i \Vdash_1 V_i : \mathbf{s}(\tau_i) \quad \Gamma \asymp \Gamma_i \\ \Gamma \Vdash_1 P : \pi\end{array}}{\Gamma \sqcap \Gamma_{1 \leq i \leq n} \Vdash_1 u!\langle\tilde{V}\rangle P : \pi} \qquad (2)\ \frac{\begin{array}{l}\Gamma(u) \leq \langle(\tilde{\tau}'), (\tilde{\tau})\rangle \\ \Gamma_i \Vdash_1 V_i : \tau_i \quad \Gamma \asymp \Gamma_i \\ \Gamma \Vdash_1 P : \mathtt{proc}\end{array}}{\Gamma \sqcap \Gamma_{1 \leq i \leq n} \Vdash_1 u!\langle\tilde{V}\rangle P : \mathtt{proc}}$$

$$\textsc{Par}_s: \ \frac{\Gamma_i \Vdash_1 P_i : \pi_i \quad \Gamma_1 \asymp \Gamma_2}{\Gamma_1 \sqcap \Gamma_2 \Vdash_1 P_1 \mid P_2 : \pi_1 \sqcap \pi_2} \qquad \textsc{Res}_s: \ \frac{\Gamma, a:\sigma \Vdash_1 P : \mathtt{proc} \quad \Gamma \vdash_1 \mathtt{SBL}}{\Gamma \Vdash_1 (\nu a:\sigma)P : \mathbf{s}(\mathtt{proc})}$$

$\textsc{Rep}$, $\textsc{Res}$, $\textsc{Spawn}_l$ from Figure 9 and $\textsc{In}_m$ from $\mathbf{Min}_{\pi\lambda}$.

Under each heading the rules must be applied in the enumerated order.

FIGURE 10. Sendable Typing Rule ($\mathbf{Sbl}_{\mathrm{D}\pi\lambda}$)

and the functional aspects of the language, as in Facile [9]. However extensions of our capability based typing systems to more advanced distributed primitives, such as hierarchical location spaces [36], process mobility [6, 33, 11], and cryptographic constructs [1, 14] will be more challenging. Since in our language we inherit the standard subtyping of the λ-calculus, it is also be possible to consider the introduction of richer subtyping relations, for example those based on records, recursive types, or polymorphic types into type systems for distributed languages.

As argued in [9, 8, 21, 25, 33], many practical applications call for parameterised higher-order process passing, which may be difficult to represent directly without functional constructions, even in languages which support migration of the processes. Moreover their presence leads to a natural and powerful programming style; an example is given in Appendix D.

It has been argued that in some sense there is no need for higher-order constructs in π-calculus based languages. For example in [31] there is a concise translation of processes using higher order values into a first order process language where only channel names are transmitted. However the translation is rather complicated and would be difficult to use as a basis for reasoning directly about the behaviour of higher-order processes. Moreover, as we will now argue in the context of $D\pi\lambda$, certain information is lost in such translations.

The basic idea of the translation is to replace the transmission of an abstraction with the transmission of a newly generated *trigger*. An application to the abstraction is then replaced by a transmission of the data to the trigger, which provides a copy of the abstraction body to process the data. Using this idea **sqServ** is replaced by

$$[\![\mathbf{sqServ}]\!] \Longleftarrow *\mathrm{req}?(r).\,(\nu\,\mathrm{tr})\,(r!\langle \mathrm{tr}\rangle \mid \mathbf{S}_{\mathrm{tr}}) \quad \text{with } \mathbf{S}_{\mathrm{tr}} \Longleftarrow *\mathrm{tr}?(x).\,\mathbf{sq}(x)$$

Here when a request is received, a new trigger is generated, and then returned to the client. Associated with the trigger is a trigger server, $\mathbf{S}_{\mathrm{tr}}$ which receives data on the trigger and then executes the body, namely $z!\langle \mathtt{sq}(x)\rangle$. Suppose we have the following $\mathbf{Client}_2$ who may already have a square server, for example for faster parallel evaluation.

$$\mathbf{Client}_2 \Longleftarrow (\nu\, ar)(\mathrm{req}!\langle r\rangle.\, r?(X).\, X\,a \mid \mathbf{sq}(a) \mid a!\langle 1, c_1\rangle \mid a!\langle 2, c_2\rangle \mid \cdots)$$

Then the client is replaced by

$$[\![\mathbf{Client}_2]\!] \Longleftarrow (\nu\, ar)(\mathrm{req}!\langle r\rangle.\, r?(\mathrm{tr}).\, \mathrm{tr}!\langle a\rangle \mid \mathbf{sq}(a) \mid a!\langle 1, c_1\rangle \mid a!\langle 2, c_2\rangle \mid \cdots)$$

The application in $\mathbf{Client}_2$ is replaced by a transmission of a to the trigger, which was received in response to the request.

However there is an essential difference between the two systems:

$$\mathbf{sqServ} \parallel \mathbf{Client}_2 \qquad\qquad [\![\mathbf{sqServ}]\!] \parallel [\![\mathbf{Client}_2]\!]$$

In the former, the new receptor $\mathbf{sq}(a)$ is created in the client location, whereas in the latter $\mathbf{sq}(a)$ is created on the server side:

$$\begin{array}{l}\mathbf{sqServ} \parallel \mathbf{Client}_2 \longrightarrow\!\!\!\!\!\rightarrow \mathbf{sqServ} \qquad\qquad \parallel \ (\nu\, a)(\mathbf{sq}(a) \mid \mathbf{sq}(a) \mid \cdots) \\ [\![\mathbf{sqServ}]\!] \parallel [\![\mathbf{Client}_2]\!] \longrightarrow\!\!\!\!\!\rightarrow (\nu\, a)(\mathbf{sq}(a) \mid [\![\mathbf{sqServ}]\!] \parallel \mathbf{sq}(a) \mid a!\langle 1, c_1\rangle \cdots)\end{array}$$

This disturbs the locality constraints on the channel a. Actually we can check that for all Γ, we have: $\Gamma \not\vdash_1 [\![\mathbf{Client}_2]\!] \parallel [\![\mathbf{sqServ}]\!]$, since a should be used as input capability in the server side to create a new $\mathbf{sq}(a)$. But $\mathbf{sqServ} \parallel \mathbf{Client}_2$ is typable, as seen in Example 4.7.

This example shows that it would be extremely difficult to adapt the translation technique in [31] so that the local typing structure is preserved; it provides at least one reason why the semantics of higher order distributed calculi are worthy of investigation, independently of their translations into first order calculi.

Preserving the locality of channels has been studied extensively for the π-calculus, [3, 22, 4, 38, 11]. For example the (untyped) *local π-calculus* [22, 4, 38] is simply defined with the following input restriction rule

$$a?(x).\,P \qquad \text{if } x \text{ does not appear as a free input subject in } P$$

If we consider the subset of $D\pi\lambda$, where the abstraction mechanisms are omitted and only a single location is used then the typing system automatically enforces this restriction on well-typed terms. However it would be wrong to generalise this restriction to higher-order terms by imposing the constraint:

$$\lambda x.P \qquad \text{if } x \text{ does not appear as a free input subject in } P$$

This is too strong; new receptors can never be created by β-reduction, hence Example 4.7 would no longer be typable. Moreover this idea does not work if we wish to control higher-order abstractions and variables, as seen in Example 3.6.

A locality condition similar to ours is used in [11] in describing various kinds of encodings in Distributed Join-Calculus. Our approach is more general; we have a formal typing system for arbitrary higher-order process passing and instantiation which ensures locality of receptors, although new receptors can be created inside the same location.

Acknowledgements

ACKNOWLEDGEMENTS: We thank James Riely, Kohei Honda and the members of Wednesday Study Group of Sussex University for comments and discussions.

A Proofs of Section 2

This appendix gives proofs of Theorem 2.6. First we need the following standard lemma.

LEMMA A.1.

(i) (substitution) *If* $\Gamma, x{:}\tau \vdash P : \rho$ *and* $\Gamma \vdash V : \tau$*. Then* $\Gamma \vdash P\{V/x\} : \rho$.

(ii) (struct) *If* $\Gamma \vdash P : \mathtt{proc}$ *and* $P \equiv Q$*, then we have* $\Gamma \vdash Q : \mathtt{proc}$.

PROOF. (1) is proved by the induction on P in the standard way. For (2), since $P \equiv Q$ is defined only between processes, the proof follows the standard manner as done in [35, 27, 15]. Hence we omit the proofs. □

Now we prove Theorem 2.6. Then by Lemma A.1 (2) above, we only have to prove the following two cases.

(1) Suppose $\Gamma \vdash (\lambda(x{:}\tau).Q)V : \rho$ and $(\lambda(x{:}\tau).Q)V \longmapsto Q\{V/x\}$. Then we have: $\Gamma \vdash Q\{V/x\} : \rho$.

(2) Suppose $\Gamma \vdash u?(\tilde{x}{:}\tilde{\tau}).P \,|\, u!\langle\tilde{V}\rangle Q : \mathtt{proc}$ and $u?(\tilde{x}{:}\tilde{\tau}).P \,|\, u!\langle\tilde{V}\rangle Q \longmapsto P\{\tilde{V}/\tilde{x}\} \,|\, Q$. Then we have: $\Gamma \vdash P\{\tilde{V}/\tilde{x}\} \,|\, Q : \mathtt{proc}$.

Case (1): Suppose $\Gamma \vdash ((\lambda(x{:}\tau).P)\,V) : \rho'$. Then we have the following derivation.

$$\frac{\Gamma, x{:}\tau \vdash_1 P : \rho_1 \quad \Gamma \vdash_1 V : \tau_2 \quad \tau_2 \leq \tau \quad \rho_1 \leq \rho'}{\Gamma \vdash (\lambda(x{:}\tau).P)\,V : \rho'}$$

Then by SUB, we have $\Gamma \vdash_1 V : \tau$. Hence by Lemma A.1 (1), we get: $\Gamma \vdash P\{V/x\} : \rho_1$. By applying SUB again, we have $\Gamma \vdash P\{V/x\} : \rho$, as desired.

Case (2): Suppose $\Gamma \vdash u?(\tilde{x}{:}\tilde{\tau}).Q \,|\, u!\langle\tilde{V}\rangle R : \mathtt{proc}$. Then we have the following derivations:

$$\Gamma \vdash u : (\tilde{\tau})^{\mathtt{I}}, \quad \text{and} \quad \Gamma,\, \tilde{x}{:}\tilde{\tau} \vdash Q : \mathtt{proc}$$

and

$$\Gamma \vdash u : (\tilde{\tau}')^{\mathtt{O}}, \quad \Gamma \vdash u : V_i{:}\tau_i', \quad \text{and} \quad \Gamma \vdash R : \mathtt{proc}$$

Let us define $\Gamma(u) = \langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$. Then by SUB, we know: $S_{\mathtt{I}} \leq (\tilde{\tau})$ and $S_{\mathtt{O}} \geq (\tilde{\tau}')$. Since always $S_{\mathtt{O}} \leq S_{\mathtt{I}}$ by definition, we know $(\tilde{\tau}') \leq (\tilde{\tau})$, which implies $\Gamma \vdash u : V_i{:}\tau_i$ by SUB. Now applying Lemma A.1 (1), we have $\Gamma \vdash P\{\tilde{V}/\tilde{x}\} \,|\, Q : \mathtt{proc}$. □

B Proofs of Section 4

B.1 *Proof of Proposition 4.5*

As in the proof of Proposition 6.2 in [15], we show the operators, $\sqcap$ and $\sqcup$ defined in the proof of Proposition 4.5 are partial meet and join operators. We only have to show, for every type α, β, γ,

(a)	$\alpha \leq \beta$ and $\alpha \leq \gamma$	imply	$\beta \sqcap \gamma$ defined	and	$\alpha \leq \beta \sqcap \gamma$
(b)	$\beta \leq \alpha$ and $\gamma \leq \alpha$	imply	$\beta \sqcup \gamma$ defined	and	$\beta \sqcup \gamma \leq \alpha$
(c)	$\beta \sqcap \gamma$	implies	$\beta \sqcap \gamma \leq \beta$		
(d)	$\beta \sqcup \gamma$	implies	$\beta \leq \beta \sqcup \gamma$		

By Lemma 4.4, we can consider every type takes a form either (1),(2) or (3) in Lemma 4.4. Only interesting cases are β is sendable but γ is not sendable. Others are easy by inductive hypothesis on types.

Base case ($k = 0$ in (2,3) of Lemma 4.4): The only interesting case is channel types. Suppose $\beta = \mathbf{s}(\langle \top, S_{1\mathtt{O}}\rangle)$ and $\gamma = \langle S_{2\mathtt{I}}, S_{2\mathtt{O}}\rangle$. For (a), suppose $\alpha \leq \beta$ and $\alpha \leq \gamma$. Then $\alpha \leq \beta$ implies $\alpha = \mathbf{s}(\langle \top, S_{2\mathtt{O}}\rangle)$ by Proposition 4.3. Then by this and $\alpha \leq \gamma$, we have $S_{2\mathtt{I}} = \top$. By induction on $S_{1\mathtt{O}}$ and $S_{2\mathtt{O}}$, $S_{1\mathtt{O}} \sqcup S_{2\mathtt{O}}$ always exists, and we have $S_{1\mathtt{O}} \sqcup S_{2\mathtt{O}} \leq S_{0\mathtt{O}}$. Hence $\beta \sqcap \gamma = \mathbf{s}(\langle \top, S_{1\mathtt{O}} \sqcup S_{2\mathtt{O}}\rangle)$ is always defined, and $\alpha \leq \beta \sqcap \gamma$. For (b), we first note that $\gamma \leq \alpha$ implies α should be non-sendable, and, by $\beta \leq \alpha$, we can set $\alpha = \langle \top, S_{2\mathtt{O}}\rangle$. Then the rest of reasoning is just similar with (a). (c) and (d) are also similar with (a) and (b), respectively.

Inductive Case ($k \geq 1$ in Lemma 4.4): The only interesting cases are β is in (1) in Lemma 4.4, while γ is in (2) or (3). Here we must check operations $\beta \sqcap \gamma$ and $\beta \sqcup \gamma$ do not induce illegal arrow types $\tau \to \rho$ such that $\rho \in \mathsf{Sble}$ but $\tau \notin \mathsf{Sble}$. We only show the case of γ in (3). The case (2) is just similar. For (a), suppose

$$\beta \simeq \mathbf{s}(\tau_{11}) \to \cdots \to \mathbf{s}(\tau_{k1}) \to \mathbf{s}(\rho_1) \quad \text{and} \quad \gamma \simeq \mathbf{s}(\tau_{12} \to \cdots \tau_{k2} \to \rho_2)$$

where $\alpha \leq \beta$ and $\alpha \leq \gamma$ for some α. First we immediately know α can not take the form (2) in Lemma 4.4 by $\alpha \leq \beta$ and Proposition 4.3. So let us define $\alpha \simeq \mathbf{s}(\tau_1 \to \cdots \to \rho)$. Then $\alpha \leq \beta$ implies $\rho \leq \mathbf{s}(\rho_1)$, hence by Proposition 4.3 again, we have $\rho \simeq \mathbf{s}(\rho')$ for some ρ'. By the formation of arrow type, we have: $\tau_i \simeq \mathbf{s}(\tau_i')$. Hence α should take a form

$$\alpha \;\simeq\; \mathbf{s}(\tau_1') \to \cdots \mathbf{s}(\tau_k') \to \mathbf{s}(\rho')$$

for some τ_i' and ρ' with $\mathbf{s}(\tau_i') \geq \mathbf{s}(\tau_{i1})$, $\mathbf{s}(\tau_i') \geq \tau_{i2}$, $\mathbf{s}(\rho') \leq \mathbf{s}(\rho_1)$, and $\mathbf{s}(\rho') \leq \rho_2$. Then we can set $\mathbf{s}(\tau_{i2}') \simeq \tau_{i2}$ for some τ_{i2}' because of $\mathbf{s}(\tau_i') \geq \tau_{i2}$ and by Proposition 4.3 again. Now by inductive hypothesis, we have $\mathbf{s}(\tau_{i1}) \sqcup \mathbf{s}(\tau_{i2}') = \mathbf{s}(\tau_{i1} \sqcup \tau_{i2}') \leq \mathbf{s}(\tau_i)$ and $\mathbf{s}(\rho_1) \sqcap \rho_2 = \mathbf{s}(\rho_1 \sqcap \rho_2) \geq \mathbf{s}(\rho)$, which implies

$$\beta \sqcap \gamma \;\simeq\; \mathbf{s}(\tau_{11} \sqcup \tau_{12}') \to \cdots \mathbf{s}(\tau_{k1} \sqcup \tau_{k2}') \to \mathbf{s}(\rho_1 \sqcap \rho_2)$$

is a well-formed arrow type, and $\alpha \leq \beta \sqcap \gamma$. Other cases are similar. □

B.2 *Proofs of Lemma 4.12*

The proof of **(1)** is obvious. For **(2)**, we use induction on the derivation of $\Gamma \vdash_1 P : \tau$. We examine the last inference used in this derivation. If this uses the rule TERM_l in Figure 9, then the result is obvious. We consider a representative sample of other cases.

Case ID: $P = u$. Take $\Delta = \{\Gamma(u)\}$.

Case SUB: Follows by induction, since Sble is downwards closed with respect to subtyping.

Case CONST: Take $\Delta = \emptyset$.

Case APP: Then there is a derivation such that

$$\frac{\Gamma \vdash_1 Q : \tau \to \rho \quad \Gamma \vdash_1 R : \tau}{\Gamma \vdash_1 QR : \rho}$$

for some τ. By the constraint on arrow types we know that $\tau \in \mathsf{Sble}$ and therefore $\tau \to \rho$ is also in Sble. This enables us to apply induction to obtain sendable environments Δ_i such that $\Gamma \leq \Delta_i$, $\Delta_1 \vdash_1 Q : \tau \to \rho$ and $\Delta_2 \vdash_1 R : \mathbf{s}(\tau)$. The required sendable environment is $\Delta = \Delta_1 \sqcap \Delta_2$ since one can easily show that $\Delta \vdash_1 QR : \tau$.

Case ABS: Suppose we have the following inference

$$\frac{\Gamma, x\!:\!\tau \vdash_1 Q : \rho}{\Gamma \vdash_1 \lambda(x\!:\!\tau).Q : \tau \to \rho}$$

where $\tau \to \rho \in \mathsf{Sble}$. Then we know $\tau,\ \rho \in \mathsf{Sble}$ and we can use induction to find a sendable environment Δ_1 such that $\Gamma, x:\tau \leq \Delta_1$ and $\Delta_1 \vdash_1 Q : \rho$. Then $\Delta_1 \sqcap \{x\!:\!\tau\} \vdash_1 Q : \rho$ from which we can derive $\Delta \vdash_1 \lambda(x\!:\!\tau).Q : \tau \to \rho$, where Δ is the sendable environment obtained by eliminating x from $\Delta_1 \sqcap \{x\!:\!\tau\}$.

(3) We only have to think the case $\rho = \mathbf{s}(\rho')$, $x \in \mathsf{fv}(P)$ and $\Gamma \vdash_1 P : \rho$ is inferred by TERM$_l$ rule. Other cases are standard. Then by Lemma 4.12 (2) above, we have the derivations such that $\Gamma_1, x\!:\!\tau' \vdash P : \rho$ for some $\Gamma \leq \Gamma_1$ and $\Gamma_1 \vdash \mathtt{SBL}$ and $\tau' \simeq \mathbf{s}(\tau_0) \geq \tau$. Hence automatically V is a sendable value. Then, by Lemma 4.12 (2) again, there is a derivation $\Gamma_2 \vdash V : \tau$ with $\Gamma_2 \vdash \mathtt{SBL}$. We then need to prove the following stronger property.

$$\Gamma_1, x\!:\!\tau \vdash P : \rho \text{ and } \Gamma_2 \vdash V : \tau \text{ with } \Gamma_1 \asymp \Gamma_2 \text{ implies } \Gamma_1 \sqcap \Gamma_2 \vdash P\{V/x\} : \rho.$$

Note for the case of $\rho = \mathbf{s}(\rho')$, we can take Γ_1 and Γ_2 as sendable environments. Since $\Gamma_1 \sqcap \Gamma_2 \geq \Gamma$ and $\Gamma_1 \sqcap \Gamma_2 \vdash \mathtt{SBL}$ by Lemma 4.12 (1), applying TERM$_l$ again, we will get the required result.

Case $P \equiv x$. Then $x\!:\!\tau \vdash_1 x\!:\!\mathbf{s}(\rho)$ with $\tau \leq \mathbf{s}(\rho)$. Then there is a derivation $\Gamma_2' \vdash V : \tau$ with $\Gamma_2' \vdash_1 \mathtt{SBL}$. By applying SUB, we have $\Gamma_2' \vdash V : \mathbf{s}(\rho)$, as required.

Case $P \equiv \lambda(y\!:\!\tau').Q$. Then we have the following derivations.

$$\frac{\dfrac{\Gamma_1, x\!:\!\tau, y\!:\!\tau' \vdash_1 Q : \rho' \quad \tau_1 \leq \tau' \quad \rho' \leq \rho_1}{\Gamma_1, x\!:\!\tau \vdash_1 \lambda(y\!:\!\tau').Q : \tau_1 \to \rho_1 \quad \Gamma_1, x\!:\!\tau \vdash \mathtt{SBL}}}{\Gamma_1, x\!:\!\tau \vdash_1 \lambda(y\!:\!\tau').Q : \mathbf{s}(\tau_1 \to \rho_1)}$$

Then by inductive hypothesis, we have:

$$\Gamma_1 \sqcap \Gamma_2, y\!:\!\tau' \vdash_1 Q\{V/x\} : \rho'$$

Then applying ABS again, we have:

$$\Gamma_1 \sqcap \Gamma_2 \vdash_1 \lambda(y\!:\!\tau').Q\{V/x\} : \tau' \to \rho'$$

By SUB and TERM$_l$, we obtain $\Gamma_1 \sqcap \Gamma_2 \vdash_1 \lambda(x\!:\!\tau').Q : \mathbf{s}(\tau_1 \to \rho_1)$, as desired.

Case $P \equiv QR$. It is easy by induction on Q and R, together with Lemma 4.12 (2).

Case $P \equiv (Q \mid R)$. Then there is a derivation such that

$$\frac{\Gamma_1, x\!:\!\tau \vdash_1 P : \mathtt{proc} \quad \Gamma_1, x\!:\!\tau \vdash_1 Q : \mathtt{proc}}{\Gamma_1, x\!:\!\tau \vdash_1 P \mid Q : \mathtt{proc}}$$

with $\Gamma_1 \vdash_1 \mathtt{SBL}$. Then by inductive hypothesis and $\Gamma_1 \sqcap \Gamma_2 \vdash \mathtt{SBL}$, we have

$$\Gamma_1 \sqcap \Gamma_2 \vdash_1 P\{V/x\} : \mathbf{s}(\mathtt{proc}) \quad \text{and} \quad \Gamma_1 \sqcap \Gamma_2 \vdash_1 Q\{V/x\} : \mathbf{s}(\mathtt{proc})$$

Then applying PAR to the above, we have done.

Case $P \equiv u!\langle\tilde{V}\rangle P, (a\!:\!\sigma)Q$ and $*Q$. Easy by inductive hypothesis on V_i, Q and R as the same as the above. □

C Type Checking Algorithms

In this subsection, we give type checking algorithms for $\pi\lambda$ and D$\pi\lambda$ using the typing rules in 5.3, combining the ideas of [24] and [35]. The predicate $\mathtt{Check}(\rho \leq \rho')$ returns true if $\rho \leq \rho'$, else returns false.[3]

TYPE CHECKING ALGORITHM FOR $\pi\lambda$ We input the following tuple (1) a term P with all bounds names/variables renamed to be distinct, (2) a union of Δ and typings $x:\tau$ of all bound names/variables in P. Then for given Δ and P, the algorithm $G(\Delta \cup \{\tilde{x}\!:\!\tilde{\tau}\}, P)$ given in Figure 11 produces a provable typing $\Delta \vdash P : \rho$. We can see if $G(\Delta,\ P) = \Delta \vdash P : \rho$, then $\Delta/(\mathsf{bv}(P) \cup \mathsf{bn}(P)) \vdash P : \rho$, and if $\Delta \vdash P : \rho$ and Δ' is typings of all bound names/variables of P, then $G(\Delta \cup \Delta', P)$ succeeds and produces the minimum type ρ' s.t. $\Delta \vdash P : \rho'$ and $\rho' \leq \rho$.

TYPE CHECKING ALGORITHM FOR D$\pi\lambda$ We construct the algorithm for the local D$\pi\lambda$, using a partial meet operator. For the local processes, we input the same tuple as G above. We presuppose all arrow types are either (1), (2) or (3) as in Section 5.3. The algorithm $G_l(\Gamma \cup \{\tilde{x}\!:\!\tilde{\tau}\},\ P)$ given in Figure 12 produces a provable typing $P : \rho$ together with a higher environment Γ such that $\Gamma \geq \Delta$. If $\Gamma \vdash_1 \mathtt{SBL}$, then term P is sendable and $\rho = \mathbf{s}(\rho')$. We can see if $G_l(\Delta,\ P) = \Delta \vdash_1 P : \rho$, then $\Delta/(\mathsf{bv}(P) \cup \mathsf{bn}(P)) \vdash P : \rho$, and if $\Gamma \vdash P : \rho$ and Γ' is a set of typings of all bound names/variables of P, then $G_l(\Gamma \cup \Gamma', P)$ succeeds.

Using G_l, an algorithm which computes $\Gamma \vdash_1 N$ (if one exits), G_s is simply constructed as defined in Figure 13 (note $\Gamma \asymp_1 \Gamma'$ is always decidable by Proposition 5.5 (2)).

[3] As discussed in 5.3, an algorithm of $\mathtt{Check}(\ )$ is constructed by extending that for subtyping in λ-calculus (cf. [20, 24]).

$$
\begin{array}{llll}
G(\Delta,\ u) & = & & \Delta \vdash u : \Delta(x) \\
G(\Delta,\ l) & = & & \Delta \vdash l : \mathtt{nat}\ \text{ etc.} \\
G(\Delta,\ PQ) & = & \text{let} & \Delta \vdash P : \tau_1 \to \rho = G(\Delta,\ P) \\
 & & & \Delta \vdash Q : \tau_2 = G(\Delta,\ Q) \\
 & & \text{in} & \text{if } \mathtt{Check}(\tau_2 \leq \tau_1) \text{ then } \Delta \vdash PQ : \rho \\
G(\Delta,\ \lambda(x{:}\tau).P) & = & \text{let} & \Delta \vdash P : \rho = G(\Delta,\ P) \\
 & & \text{in} & \Delta \vdash \lambda(x{:}\tau).P : \tau \to \rho \ \text{ with } x{:}\tau \in \Delta \\
G(\Delta,\ \mathbf{0}) & = & & \Delta \vdash \mathbf{0} : \mathtt{proc} \\
G(\Delta,\ P\,|\,Q) & = & \text{let} & \Delta \vdash P : \mathtt{proc} = G(\Delta,\ P) \\
 & & & \Delta \vdash Q : \mathtt{proc} = G(\Delta,\ Q) \\
 & & \text{in} & \Delta \vdash P\,|\,Q : \mathtt{proc} \\
G(\Delta,\ {*P}) & = & \text{let} & \Delta \vdash P : \mathtt{proc} = G(\Delta,\ P) \\
 & & \text{in} & \Delta \vdash {*P} : \mathtt{proc} \\
G(\Delta,\ (\nu a{:}\sigma)P) & = & \text{let} & \Delta \vdash P : \mathtt{proc} = G(\Delta,\ P) \\
 & & \text{in} & \Delta \vdash (\nu a{:}\sigma)P : \mathtt{proc} \ \text{ with } a{:}\sigma \in \Delta \\
G(\Delta,\ u?(\tilde{x}{:}\tilde{\tau}).P) & = & \text{let} & \Delta \vdash P : \mathtt{proc} = G(\Delta,\ P) \\
 & & \text{in} & \text{if } \mathtt{Check}(\Delta(u) \leq (\tilde{\tau})^{\mathtt{I}}) \text{ then } \Delta \vdash u?(\tilde{x}{:}\tilde{\tau}).P : \mathtt{proc} \\
G(\Delta,\ u!\langle \tilde{V} \rangle P) & = & \text{let} & \Delta \vdash P : \mathtt{proc} = G(\Delta,\ P) \\
 & & & \Delta \vdash V_i : \tau_i \\
 & & \text{in} & \text{if } \mathtt{Check}(\Delta(u) \leq (\tilde{\tau})^{\mathtt{O}}) \text{ then } \Delta \vdash u!\langle \tilde{V} \rangle P : \mathtt{proc}
\end{array}
$$

FIGURE 11. Algorithm for $\pi\lambda$

D Example of higher-order programming: Distributed Databases

In this appendix, we show that the ability to transmit higher-order abstractions provides a powerful programming mechanism for distributed systems.

First we introduce a simple agent, called a *switcher*, which sends a value V to the received channel z.

$$\mathbf{sw}(aV) \Longleftarrow {*a?(z).\ z!\langle V \rangle}$$

Suppose the client (A) wants to ask for information of books to the nearest bookstore (B); for example he might only know the title of the first book ("title$_1$"), and the author ("author$_2$") of the second. (A) sends his script as a program abstraction $\lambda(A,D,T).\ (AD(\text{title}_1), DT(\text{author}_2))$ with the continuation a to (B). Then (B) automatically applies the fixed several services (here query-functions **?author**, **?date** and **?title**) to the requested information.

$$\mathbf{Client}_A \Longleftarrow b!\langle \lambda(A,D,T).\ (AD(\text{title}_1), DT(\text{author}_2)), a \rangle$$

$$\mathbf{BStore}_B \Longleftarrow {*b?(Y,z).\ (\mathtt{let}\ y = Y\ (\mathbf{?author}, \mathbf{?date}, \mathbf{?title})\ \mathtt{in}\ c!\langle\, \lambda(X).(X\,y),\ z \rangle)}$$

$$
\begin{array}{llll}
G_l(\Delta,\ x) & = & & \Delta; x{:}\Delta(x) \vdash_1 x : \Delta(x) \\
G_l(\Delta,\ l) & = & & \Delta; \emptyset \vdash_1 l : \mathbf{s}(\mathtt{nat}) \\
G_l(\Delta,\ a) & = & & \Delta; a{:}\langle \top, S_{\mathtt{O}} \rangle \vdash_1 a : \mathbf{s}(\langle \top, S_{\mathtt{O}} \rangle) \text{ with } \Delta(a) = \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \\
 & & \text{or} & \Delta; a{:}\Delta(a) \vdash_1 a{:}\Delta(a) \\
G_l(\Delta,\ PQ) & = & \text{let} & \Delta; \Gamma_1 \vdash_1 P : \tau_1 = G_l(\Delta,\ P) \\
 & & & \Delta; \Gamma_2 \vdash_1 Q : \tau_2 = G_l(\Delta,\ Q) \\
 & & & \quad \text{with } \tau_1 = \mathbf{s}(\tau_0 \to \rho) \text{ or } \tau_1 = \tau_0 \to \rho \\
 & & \text{in} & \text{if } \mathtt{Check}(\tau_0 \geq \tau_2) \text{ then } \Delta; \Gamma_1 \sqcap \Gamma_2 \vdash_1 PQ : \rho \\
G_l(\Delta,\ \lambda(x{:}\tau).P) & = & \text{let} & \Delta; \Gamma \vdash_1 P : \rho = G_l(\Delta,\ P) \\
 & & & \tau = \Gamma(x) \\
 & & \text{in} & \text{if } \rho = \mathbf{s}(\rho_0) \wedge \tau = \mathbf{s}(\tau_0) \\
 & & & \text{then } \Delta; \Gamma/x \vdash_1 \lambda(x{:}\tau).P : \mathbf{s}(\tau \to \rho) \\
 & & & \text{else if } \Gamma/x \vdash_1 \mathtt{SBL} \text{ then } \Delta; \Gamma/x \vdash_1 \lambda(x{:}\tau).P : \mathbf{s}(\tau \to \mathtt{up}(\rho)) \\
 & & & \text{else } \Delta; \Gamma/x \vdash_1 \lambda(x{:}\tau).P : \tau \to \mathtt{up}(\rho) \\
G_l(\Delta,\ \mathbf{0}) & = & & \Delta; \emptyset \vdash_1 \mathbf{0} : \mathbf{s}(\mathtt{proc}) \\
G_l(\Delta,\ P\,|\,Q) & = & \text{let} & \Delta; \Gamma_1 \vdash_1 P : \pi_1 = G_l(\Delta, P) \\
 & & & \Delta; \Gamma_2 \vdash_1 Q : \pi_2 = G_l(\Delta,\ Q) \\
 & & \text{in} & \Delta; \Gamma_1 \sqcap \Gamma_2 \vdash_1 P\,|\,Q : \pi_1 \sqcap \pi_2 \\
G_l(\Delta,\ {*P}) & = & \text{let} & \Delta; \Gamma \vdash_1 P : \pi = G_l(\Delta,\ P) \\
 & & \text{in} & \Delta; \Gamma \vdash_1 {*P} : \pi \\
G_l(\Delta,\ (\nu a{:}\sigma)P) & = & \text{let} & \Delta; \Gamma \vdash_1 P : \pi = G_l(\Delta; \Gamma,\ P) \\
 & & & \sigma = \Delta(a) \\
 & & \text{in} & \text{if } \Gamma/a = \mathtt{SBL} \text{ then } \Delta; \Gamma/a \vdash_1 (\nu a{:}\sigma)P : \mathbf{s}(\mathtt{proc}) \\
 & & & \text{else } \Delta; \Gamma/a \vdash_1 (\nu a{:}\sigma)P : \mathtt{proc}. \\
G_l(\Delta,\ u?(\tilde{x}{:}\tilde{\tau}).\ P) & = & \text{let} & \Delta \vdash_1 P : \pi = G_l(\Delta,\ P) \\
 & & \text{in} & \text{if } \mathtt{Check}(\Delta(u) \leq (\tilde{\tau})^{\mathtt{I}}) \\
 & & & \text{then } \Delta; \Gamma/x \sqcap \{u{:}(\tilde{\tau})^{\mathtt{I}}\} \vdash_1 u?(\tilde{x}{:}\tilde{\tau}).\ P : \mathtt{proc} \\
G_l(\Delta,\ u!\langle \tilde{V} \rangle.\ P) & = & \text{let} & \Delta; \Gamma \vdash_1 P : \pi = G_l(\Delta,\ P) \\
 & & & \Delta; \Gamma' \vdash_1 V_i : \tau \\
 & & \text{in} & \text{if } \tau_i = \mathbf{s}(\tau_i') \wedge \mathtt{Check}(\Delta(u) \leq (\tilde{\tau})^{\mathtt{O}}) \\
 & & & \text{then } \Delta; \Gamma \sqcap \Gamma' \sqcap \{u{:}(\tilde{\tau})^{\mathtt{O}}\} \vdash_1 u!\langle \tilde{V} \rangle.\ P : \pi \\
 & & & \text{else let } \Delta(u) = \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \\
 & & & \quad \text{in if } S_{\mathtt{I}} \neq \top \wedge \mathtt{Check}(\Delta(u) \leq (\tilde{\tau})^{\mathtt{O}}) \\
 & & & \quad \text{then } \Delta; \Gamma \sqcap \Gamma' \sqcap \{u{:}\langle S_{\mathtt{I}}, (\tilde{\tau}) \rangle\} \vdash_1 u!\langle \tilde{V} \rangle.\ P : \mathtt{proc}
\end{array}
$$

FIGURE 12. Algorithm for terms in Local $\mathrm{D}\pi\lambda$

If the result is evaluated (to a value) by (B), and then sent the to the publisher (C) to ask whether the requested books are available or not. Finally (C) puts the completed information in the switcher of the shared database (D) which is made

$$
\begin{array}{lcll}
G_s(\Delta,\, P) & = & \text{let} & \Delta;\Gamma \vdash_1 P:\pi = G_l(\Delta,\, P) \\
 & & \text{in} & \Gamma \vdash_1 P \\
G_s(\Delta,\, N_1 \,\|\, N_2) & = & \text{let} & \Gamma_1 \vdash_1 N_1 = G_s(\Delta,\, N_1) \\
 & & & \Gamma_2 \vdash_1 N_2 = G_s(\Delta,\, N_2) \\
 & & \text{in} & \text{if } \mathtt{Check}(\Gamma_1 \asymp_1 \Gamma_2) \text{ then } \Gamma_1 \sqcap \Gamma_2 \vdash_1 N_1 \,\|\, N_2 \\
G_s(\Delta,\, (a{:}\sigma)N) & = & \text{let} & \Gamma \vdash_1 N = G_s(\Delta,\, N) \\
 & & \text{in} & \Gamma/a \vdash_1 (a{:}\sigma)N \text{ with } a{:}\sigma \in \Delta
\end{array}
$$

FIGURE 13. Algorithm for systems in Local Dπλ

accessible to the client (A) at at d'.

$$\mathbf{Publisher}_C \Longleftarrow *c?(Z,z).\,\mathtt{let}\ y = (Z\,\mathbf{?exist})\ \mathtt{in}\ d!\langle y,z\rangle$$

$$\mathbf{SharedDB}_D \Longleftarrow *d?(y,z).\,(\nu\, d')(z!\langle d'\rangle \mid \mathrm{sw}(d'y))$$

Then we compose these systems in the following way.

$$\mathbf{Client}_A \,\|\, (\nu\, cd)(\mathbf{BStore}_B \,\|\, \mathbf{Publisher}_C \,\|\, \mathbf{SharedDB}_D)$$

We note:

(1) sending the script (the program abstraction) prepared by the client side is essential to apply the fixed and incremental services (here query-functions) available by each server (here (B) and (C)),

(2) the let expression, available because of the call-by-value evaluation strategy, guarantees the partial evaluation of queries and subsequent transmission to the next server, and

(3) (A) does not need to know the publisher's name c and the shared database name d, but it can be offered a series of services from (C) and (D); the necessity of hiding such communication points from the outside is often found in real distributed database systems, often for security reason.

More generally, we can describe distributed service systems in Dπλ using the following schema:

$$
\begin{array}{ll}
 & \mathbf{Client}_A \Longleftarrow b_1!\langle \lambda(x_1).\,(x_1(V_1), x_1(V_2), \ldots), c\rangle \\
\| & \mathbf{Service}_1 \Longleftarrow *b_1?(Y,z).\,\mathtt{let}\ y = (Y\ f_1)\ \mathtt{in}\ b_2!\langle\, \lambda(x_2).(x_2\, y), z\rangle \\
\| & \mathbf{Service}_2 \Longleftarrow *b_2?(Y,z).\,\mathtt{let}\ y = (Y\ f_2)\ \mathtt{in}\ b_3!\langle\, \lambda(x_3).(x_3\, y), z\rangle \\
\ldots & \\
\| & \mathbf{Service}_n \Longleftarrow *b_n?(Y,z).\,\mathtt{let}\ y = (Y\ f_n)\ \mathtt{in}\ d!\langle y, d\rangle
\end{array}
$$

Finally **Service**$_n$ sends the final information to the shared database (D) to store and possibly make it selectively available.

References

[1] Abadi, M. and Gordon, A., The Spi-calculus, Computer and Communications Security, pp.36–47, ACM Press, 1997.

[2] Amadio, R., Translating Core Facile, ECRC Research Report 944-3, 1994.

[3] Amadio, R., An asynchronous model of locality, failure, and process mobility. INRIA Report 3109, 1997.

[4] Boreale, M., On the Expressiveness of Internal Mobility in Name-Passing Calculi, *Proc. CONCUR'96*, LNCS 1119, pp.163–178, Springer-Verlag, 1996.

[5] Boudol, G., The π-Calculus in Direct Style, *POPL'98*, pp.228–241, ACM Press, 1998.

[6] Cardelli, L. and Gordon, A., Typed Mobile Ambients, *Proc. POPL'99*, ACM Press, pp.79–92, 1999.

[7] Chien, A., Concurrent Aggregates, MIT Press, 1993.

[8] Ferreira, W., Hennessy, M. and Jeffrey, M., A Theory of Weak Bisimulation for Core CML, *Proc. ICFP*, pp.201–212, ACM Press, 1996. The full version appeared in *J. Func. Pro.*, 8(5):447–491,1998.

[9] Giacalone, A., Mistra, P. and Prasad, S., Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming, *Proc. ICALP'90*, LNCS 443, pp.765–780, Springer-Verlag, 1990.

[10] Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.

[11] Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., and Rémy, D., A Calculus for Mobile Agents, *CONCUR'96*, LNCS 1119, pp.406–421, Springer-Verlag, 1996.

[12] Sun Microsystems Inc., Java home page. http://www.javasoft.com/, 1995.

[13] Hartonas, C. and Hennessy, M., Full Abstractness for a Functional/Concurrent Language With Higher-Order Value-Passing, *Information and Computation*, Vol. 145, pp.64–106, 1998.

[14] Heintze, N. and Riecke, J., The SLam Calculus: Programming with Secrecy and Integrity, *Proc. POPL'98*, pp.365-377. ACM Press, 1998.

[15] Hennessy, M. and Riely, J., Resource Access Control in Systems of Mobile Agents, CS Report 02/98, University of Sussex, http://www.cogs.susx.ac.uk, 1998.

[16] Honda, K., Composing Processes, *POPL'96*, pp.344-357, ACM Press, 1996.

[17] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. *ECOOP'91*, LNCS 512, pp.133–147, Springer-Verlag 1991.

[18] Honda, K. and Yoshida, N., On Reduction-Based Process Semantics. *TCS*, pp.437–486, No.151, North-Holland, 1995.

[19] Jeffrey, A. and Wakeman, I., SafetyNet: Programming Active Networks, Available from: http://klee.cs.depaul.edu/an/, 1998.

[20] Jim, T. and Palsberg, J., Type Inference in Systems of Recursive Types with Subtyping, pp.23, July, 1997. Available at: http://www.cis.upenn.edu/˜ tjim,

[21] Leth, L. and Thomsen, B., Some Facile Chemistry, ERCC Technical Report, ERCC-92-14, 1992.

[22] Merro, M. and Sangiorgi, D., On asynchrony in name-passing calculi, *ICALP'98*, LNCS 1443, pp.856–867, Springer-Verlag, 1998.

[23] Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes. *Information and Computation*, 100(1), pp.1–77, 1992.

[24] Mitchell, J., Type inference with simple subtypes. *J. Functional Programming*, 1(3):245–285, July, 1991.

[25] De Nicola, R., Ferrari, G. and Pugliese, R., Klaim: a Kernel Language for Agents Interaction and Mobility IEEE Trans. on Software Engineering, Vol.24(5), May, 1998.

[26] O'Hearn, P., Power, J., Takeyama, M., and Tennent, D., Syntactic Control of Interference Revised, *Proc. MFPS'97*, *ENCS*, Elsevier, 1997.

[27] Pierce, B.C. and Sangiorgi. D, Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.

[28] Pierce, B. and Turner, D., Pict: A Programming Language Based on the Pi-calculus, Indiana University, CSCI Technical Report, 476, March, 1997.

[29] Plotkin, G., Call-by-name, call-by-value and the lambda-calculus, *TCS*, 1:125–159, 1975.

[30] Riely, J. and Hennessy, M., Trust and Partial Typing in Open Systems of Mobile Agents, CS Technical Report, 04/98, Available at: http://www.cogs.susx.ac.uk, 1998.

[31] Sangiorgi, D., *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms*. Ph.D. Thesis, University of Edinburgh, 1992.

[32] Sangiorgi, D., The name discipline of uniform receptiveness, *Proc. ICALP'97*, LNCS 1256, pp.303–313, 1997.

[33] Sekiguchi, T. and Yonezawa, A., A Calculus with Code Mobility, Proc. IFIP, pp.21–36, Chapman & Hall, 1997.

[34] Sewell, P., Global/Local Subtyping and Capability Inference for a Distributed π-calculus, *ICALP'98*, LNCS 1443, pp.695–706, Springer-Verlag, 1998.

[35] Vasconcelos, V. and Honda, K., Principal Typing Scheme for Polyadic π-Calculus. *CONCUR'93*, LNCS 715, pp.524-538, Springer-Verlag, 1993.

[36] Vitek, J. and Castagna, G., A Calculus of Secure Mobile Computations, Available at: http://cuiwww.unige.ch/˜ jvitek, 1999.

[37] Yoshida, N., Graph Types for Monadic Mobile Processes, *FST/TCS'16*, LNCS 1180, pp. 371–386, Springer-Verlag, 1996. Full version as LFCS Technical Report, ECS-LFCS-96-350, 1996.

[38] Yoshida, N., Minimality and Separation Results on Asynchronous Mobile Processes: Representability Theorems by Concurrent Combinators. Proc. CONCUR'98, pp.131–146, LNCS 1466, Springer-Verlag, 1998. Full version available as CS Report 05/98, University of Sussex, 1998. Available at: http://www.cogs.susx.ac.uk, 1998.