Subtyping and Locality in Distributed Higher Order Processes

NOBUKO YOSHIDA MATTHEW HENNESSY

e-mail: nobuko@cogs.susx.ac.uk
matthewh@cogs.susx.ac.uk

School of Cognitive and Computing Sciences
Brighton, United Kingdom, BN1 9QH

tel:+44-1273-678-763
fax:+44-1273-671-320

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 a more general channel constraints studied by Sewell in a higher-order distributed setting.

Contents

Subtyping and Locality in Distributed Higher Order Processes

NOBUKO YOSHIDA MATTHEW HENNESSY

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, 21, 24, 29]. Most of these are based on first-order extensions of the π-calculus [22]; 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 [26]. However these translations, as we will explain in 6.1, 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 [25] and the π-calculus [22], 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 with the core version of Facile [2, 10, 20] and CML [9] and can be regarded as an extension of Blue-Calculus [6] to a higher-order term passing.

A desirable feature of some distributed systems is that every channel name is associated with a *unique* receptor site, which is called *receptiveness* in [27]; another property called *locality* where new receptors are not created by received channels, has also been studied in [3, 5, 21, 32] 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 in, e.g. CONCURRENT AGGREGATES [8]; 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 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 [23, 16] to a higher-order setting where the order theoretic property of subtyping relation

[0]Full version available at: `http://www.cogs.susx.ac.uk/users/nobuko/index.html`.

$$
\begin{array}{lrcl}
\text{Term:} & P,Q,\ldots \in \mathit{Term} & ::= & V \mid \mathtt{let}\, x{:}\tau_1 = P\, \mathtt{in}\, Q \mid P\,Q \\
& & \mid & u?(\tilde{x}{:}\tilde{\tau}).P \mid u!\langle \tilde{V}\rangle P \mid (\nu a{:}\sigma)P \mid *P \mid P \mid Q \mid \mathbf{0} \\
\text{Value:} & V,W,\ldots \in \mathit{Val} & ::= & u \mid \lambda(x{:}\tau).P \\
\text{Identifier:} & u,v,\ldots \in \mathit{Id} & ::= & l \mid a \mid x \\
\text{Literal:} & l,l',\ldots \in \mathit{Lit} & ::= & \mathtt{true} \mid \mathtt{false} \mid () \mid 0 \mid 1 \mid \ldots
\end{array}
$$

FIGURE 1. Syntax of $\pi\lambda$

plays a pivotal role for a natural integration with arrow types. 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 Sewell [29] in a higher-order setting, and the proof of a multiple higher-order replication theorem extended from [23, 27]. Section 6 concludes with discussion and related work. Due to space limitation, we leave all proofs and detailed definitions to the full version [33].

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 [25] augmented with the π-calculus primitives [22]. 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 oder 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. In the former there are values, consisting of basic values and abstractions, together with application and a form of *let* construct. 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 $\mathtt{fn}(P)/\mathtt{bn}(P)$ and $\mathtt{fv}(P)/\mathtt{bv}(P)$ 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.

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 $\longrightarrow\!\!\!\!\!\rightarrow$ to denote multi-step reductions.

Reduction Rules:

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

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

$$(\text{let}_1)\ \frac{P \longrightarrow P'}{\texttt{let}\, x{:}\tau = P\ \texttt{in}\ Q \longrightarrow \texttt{let}\, x{:}\tau = P'\ \texttt{in}\ Q} \qquad (\text{app}_l)\ \frac{P \longrightarrow P'}{P\,Q \longrightarrow P'\,Q}$$

$$(\text{let}_2)\ \texttt{let}\, x{:}\tau = V\ \texttt{in}\ Q \longrightarrow Q\{V/x\} \qquad (\text{app}_r)\ \frac{Q \longrightarrow Q'}{V\,Q \longrightarrow V\,Q'}$$

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

Structure Equivalence:

- $P \equiv Q$ if $P \equiv_\alpha Q$.
- $P \,|\, Q \equiv Q \,|\, P \quad (P \,|\, Q) \,|\, R \equiv P \,|\, (Q \,|\, R) \quad P \,|\, \mathbf{0} \equiv P \quad *P \equiv P \,|\, *P$
- $(\nu\, a)\mathbf{0} \equiv \mathbf{0} \quad (\nu\, a)(\nu\, b)P \equiv (\nu\, b)(\nu\, a)P \quad (\nu\, a)P \,|\, Q \equiv (\nu\, a)(P \,|\, Q)$ if $a \notin \mathtt{fn}(Q)$

FIGURE 2. Reduction for $\pi\lambda$

EXAMPLE 2.1. (sq-server) 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 fact 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 then returns the processed data along the return channel. Then 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:

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

□

IO-TYPES We use as types for $\pi\lambda$ a simplification of the input/output capabilities of [16] (in turn a *strict* generalisation of [23]). They are defined in Figure 3, where we assume a given set of base types, such as `nat` 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 $\rightarrow$, as in the λ-calculus. However here in addition we may also use channel types, ranged over by σ. These take the form $\langle S_{\mathrm{I}}, S_{\mathrm{O}}\rangle$, a pair consisting of an *input sort* S_{I} and an *output sort* S_{O}; these input/output sorts are in turn either a vector of value types or $\top$, denoting the highest capability, or $-$, denoting the lowest. The representation of IO-types as a tuple [17, 16] makes the definition of the subtyping relationship, also given in Figure 3, more natural when we integrate with arrow types of the λ-

Type:

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

Ordering:

(base)	$\mathtt{proc} \leq \mathtt{proc}$, $\mathtt{nat} \leq \mathtt{nat}$, $S \leq S$, etc.
$(-, \top)$	$- \leq S \quad S \leq \top$
(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)	$\sigma_i = \langle S_{i\mathtt{I}}, S_{i\mathtt{O}} \rangle,\ S_{1\mathtt{I}} \leq S_{2\mathtt{I}},\ S_{1\mathtt{O}} \geq S_{2\mathtt{O}} \;\Rightarrow\; \sigma_1 \leq \sigma_2.$

Abbreviations:

(input only)
$(\tilde{\tau})^{\mathtt{I}} \stackrel{\text{def}}{=} \langle (\tilde{\tau}), - \rangle$
(output only)
$(\tilde{\tau})^{\mathtt{O}} \stackrel{\text{def}}{=} \langle \top, (\tilde{\tau}) \rangle$
(input/output)
$(\tilde{\tau})^{\mathtt{IO}} \stackrel{\text{def}}{=} \langle (\tilde{\tau}), (\tilde{\tau}) \rangle$

FIGURE 3. Types for $\pi\lambda$

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 always takes fewer capabilities than specified by the outside environment, while a sender always send more capabilities than specified. Then, as already discussed in [16], IO-types in [23] are represented as a special case of our IO-types;[1] to denote them, we introduce the abbreviations in Figure 3. Note that $(\tilde{\tau})^{\mathtt{IO}} \leq (\tilde{\tau})^{\mathtt{I}} \leq \langle -, \top \rangle$ and $(\tilde{\tau})^{\mathtt{IO}} \leq (\tilde{\tau})^{\mathtt{O}} \leq \langle -, \top \rangle$. Note also $\langle -, \top \rangle \neq \top$ 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. The subtyping relation over types defined in Figure 3 is partial order and finite bounded complete, FBC, (cf. [16]). The partial meet operator $\sqcap$ and join operator $\sqcup$ can be also defined directly following [16]. For the base and arrow types, we define $\sqcap$ and $\sqcup$ as the standard join and meet operators w.r.t. $\leq$. For channel types, we use the following definition:

(vec)	$(\tilde{\tau}) \sqcup (\tilde{\tau}') \stackrel{\text{def}}{=} (\tilde{\tau}'')$ with $\tau''_i = \tau_i \sqcup \tau'_i$ and $(\tilde{\tau}) \sqcap (\tilde{\tau}') \stackrel{\text{def}}{=} (\tilde{\tau}'')$ with $\tau''_i = \tau_i \sqcap \tau'_i$
(chan)	(a) $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \sqcup \langle S'_{\mathtt{I}}, S'_{\mathtt{O}} \rangle \stackrel{\text{def}}{=} \langle S_{\mathtt{I}} \sqcup S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcap S'_{\mathtt{O}} \rangle$ and
	(b) $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \sqcap \langle S'_{\mathtt{I}}, S'_{\mathtt{O}} \rangle \stackrel{\text{def}}{=} \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}} \rangle$ if $S_{\mathtt{I}} \geq S'_{\mathtt{O}}$ and $S'_{\mathtt{I}} \geq S_{\mathtt{O}}$; else undefined.

If $S \sqcap S'$ (resp. $S \sqcup S'$) is undefined in (vec), (i.e. they are structually dissimilar or do not satisfy the IO constraint), then we set $S \sqcap S' = -$ (resp. $S \sqcup S' = \top$) in (a) in (chan).

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:

- $\mathtt{dom}(\Gamma)$ 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 $u \notin \mathtt{dom}(\Gamma)$.
- $\Delta \leq \Gamma$ means $\forall\, u \in \mathtt{dom}(\Gamma).\, \Delta(u) \leq \Gamma(u)$.

Then we define:

- $\Gamma \sqcap \Delta \stackrel{\text{def}}{=} \Gamma/A \cup \Delta/A \cup \{u{:}(\Delta(u) \sqcap \Gamma(u)) \mid u \in \mathtt{dom}(\Gamma) \cap \mathtt{dom}(\Delta)\}$, and
- $\Gamma \sqcup \Delta \stackrel{\text{def}}{=} \{u{:}(\Delta(u) \sqcup \Gamma(u)) \mid u \in \mathtt{dom}(\Gamma) \cap \mathtt{dom}(\Delta)\}$

[1] Our general form of IO-types, where input and output capabilities on a channel may be different [17, 16], gives us more typable terms than [23] even if we restrict the syntax to the pure polyadic π-calculus. See Example 2.5 in [33].

Common Typing Rules:

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

Functional Typing Rules:

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

$$\textsc{Let}: \quad \frac{\Gamma \vdash P : \tau \quad \Gamma, x{:}\tau \vdash Q : \rho}{\Gamma \vdash \mathtt{let}\, x{:}\tau = P \,\mathtt{in}\, Q : \rho} \qquad \textsc{App}: \quad \frac{\Gamma \vdash P : \tau \to \rho \quad \Gamma \vdash Q : \tau}{\Gamma \vdash PQ : \rho}$$

Process Typing Rules:

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

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

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

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

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 function Γ using the 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 [23, 16].

EXAMPLE 2.2. (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:

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

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

This simple typing system satisfies the following standard subject reduction theorem.

THEOREM 2.3. (Subject Reduction)

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

Syntax:

$$\begin{array}{lllll} \text{Term:} & P,Q,\ldots \in \mathit{Term} & ::= & \mathtt{Spawn}(P) \mid \cdots & \text{from Figure 1} \\ \text{System:} & M,N,\ldots \in \mathit{System} & ::= & P \mid N \,||\, M \mid (\nu a{:}\sigma)N & \end{array}$$

Others from Figure 1.

Distributed Reduction Rules:

$$\begin{array}{ll} \text{(spawn)} & (\cdots Q \,|\, \mathtt{Spawn}(P)) \longrightarrow (\cdots Q) \,||\, P \\ (\mathrm{com}_s) & (u?(\tilde{x}{:}\tilde{\tau}).P \,|\, \cdots) \,||\, (u!\langle \tilde{V}\rangle Q \,|\, \cdots) \longrightarrow (P\{\tilde{V}/\tilde{x}\} \,|\, \cdots) \,||\, (Q \,|\, \cdots) \end{array}$$

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

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

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 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 [7, 16, 29], 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 to study the use of our locality typing system in the distributed setting. The reduction semantics of the previous section is extended to the new language, D$\pi\lambda$, in a straightforward manner, outlined in Figure 5. The structural equivalence of systems is 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,

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

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 $P \downarrow a^{\mathrm{I}}$ which is satisfied when P can immediately perform input on name a. Now let us say a channel type σ is *local* if σ has an input capability, i.e $\sigma = \langle(\tilde{\tau}), S_0\rangle$. We also call a channel u is *local under* Γ if $\Gamma(u)$ is local.

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 $u{:}\langle S_{i\mathrm{I}}, S_{i0}\rangle \in \Gamma_i$ $(i = 1,2)$ implies $S_{1\mathrm{I}} = \top$ or $S_{2\mathrm{I}} = \top$. □

Intuitively this means that if a channel a is local in Γ_1, then it must not be local in another

Input Predicate:

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

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

Locality Error:

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

FIGURE 6. Locality Error

Local Distributed Rules:

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

FIGURE 7. Local Distributed Typing Rules

environment Γ_2.

The typing system for the system N, which is given by Distributed Typing Rules in Figure 7, is simply in form of $\Gamma \vdash N$ where Γ is again the same typing function. 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 $\Gamma_1 \asymp_1 \Gamma_2$ and N_i is typable with respect to Γ_i. If the term is system composable, then we have no immediate locality error since $P \downarrow a^{\mathrm{I}}$ and $\Gamma \vdash_1 P : \mathtt{proc}$ imply $\Gamma \vdash a : (\tilde{\tau})^{\mathrm{I}}$ for some $\tilde{\tau}$. That is:

THEOREM 3.2. (Type Safety) $\Gamma \vdash_1 N$ *implies* $N \not\xrightarrow{lerr}$.

It is however easy to see that the system composability as defined above is not closed under reduction: indeed, we easily have $N \not\xrightarrow{lerr}$ and $N \longrightarrow\!\!\!\!\!\rightarrow N'$ does *not* imply $N' \not\xrightarrow{lerr}$.

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. Take $a?(x).\,P \,|\, b!\langle a\rangle \;\|\; b?(y).\,y?(z).\,Q$. Then it is easy to check that this can be typed with PAR$_l$ in Figure 7. However after one reduction step, the communication along b, we obtain $a?(x).\,P \;\|\; a?(z).\,Q$, which is no longer typable. It is not difficult to exclude such terms by a simple syntactic condition or typing systems, as has been studied in [2, 27, 5, 21]. However the presence of higher-order processes makes the situation more complicated, as the following example shows.

EXAMPLE 3.3. 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

Types:

Term Type:	ρ	$::=$	$\pi \mid \tau$
Process Type:	π	$::=$	$\mathtt{proc} \mid \mathbf{s}(\mathtt{proc})$
Value Type:	τ	$::=$	$\mathtt{unit} \mid \mathtt{nat} \mid \mathtt{bool} \mid \sigma \mid \mathbf{s}(\tau)$ with $\tau \neq \sigma$
		$\mid$	$\tau \to \rho$ with $\rho \simeq \mathbf{s}(\rho') \Rightarrow \tau \simeq \mathbf{s}(\tau')$
Channel Type:	σ	$::=$	$\mathbf{s}(\langle \top, S_0 \rangle) \mid \langle S_{\mathrm{I}}, S_0 \rangle$ with $S_{\mathrm{I}} \geq S_0$, $S_{\mathrm{I}} \neq -$ and $S_0 \neq \top$.
Sort Type:	S	$::=$	as in Figure 3.

Ordering: All rules from Figure 3 and

(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)	$\mathbf{s}(\tau) \to \mathbf{s}(\rho) \leq \mathbf{s}(\mathbf{s}(\tau) \to \mathbf{s}(\rho))$
($\simeq$)	$\rho \leq \rho' \wedge \rho' \leq \rho \Rightarrow \rho \simeq \rho'$		

FIGURE 8. Locality types for D$\pi\lambda$

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

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

Then this does not destroy locality. □

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

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

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:

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

However, the following violates locality.

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

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 occurrence of $b!\langle V\rangle$, where the term V can be exported to a new 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 system is 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.

Send Rules:

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

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

Common Rules: as in Figure 4.

Functional Rules: as in Figure 4 adding a condition $\rho\simeq\mathbf{s}(\rho')\ \Rightarrow\ \tau\simeq\mathbf{s}(\tau')$ for LET.

Process Rules:

$$\textsc{Out}_d:\ \frac{\Gamma\vdash_{\mathtt{l}} u:(\mathbf{s}(\tilde{\tau}))^{\mathtt{0}}\quad\Gamma\vdash_{\mathtt{l}} V_i:\mathbf{s}(\tau_i)\quad\Gamma\vdash_{\mathtt{l}} P:\pi}{\Gamma\vdash_{\mathtt{l}} u!\langle\tilde{V}\rangle P:\pi}\qquad \textsc{Out}_l:\ \frac{\Gamma\vdash_{\mathtt{l}} u:\langle(\tilde{\tau}'),(\tilde{\tau})\rangle\quad\Gamma\vdash_{\mathtt{l}} V_i:\tau_i\quad\Gamma\vdash_{\mathtt{l}} P:\mathtt{proc}}{\Gamma\vdash_{\mathtt{l}} u!\langle\tilde{V}\rangle P:\mathtt{proc}}$$

NIL,REP,PAR,RES as in Figure 4 with `proc` replaced by π, and IN as 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 $\vdash_{\mathtt{l}}$ in INTRO.

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

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. A channel type with the input capability is not sendable. The side condition of arrow types simply avoids a sendable term having a non-sendable subterm; e.g. if either P or Q is non-sendable, then $P\,Q$ is automatically non-sendable. The first extra ordering rule says that the constructor $\mathbf{s}(\ )$ preserves subtyping; the second that all sendable values are values. In conjunction with the third, the second implies that sendability is idempotent, $\mathbf{s}(\mathbf{s}(\rho))\simeq\mathbf{s}(\rho)$. Similarly with the forth, we have: $\mathbf{s}(\mathbf{s}(\tau)\to\mathbf{s}(\rho))\simeq\mathbf{s}(\tau)\to\mathbf{s}(\rho)$.

The new type inference system is given in Figure 9. The **Send Rules** determine which values can be sent between locations. All constants and output capabilities on channels are automatically sendable. In the crucial rule $\textsc{Term}_l$ we use the notation $\Delta\vdash_{\mathtt{l}}\mathtt{SBL}$ to denote that Δ is a *sendable environment*, that is it consists only of sendable types; formally if $u:\tau\in\Delta$, then (1) $\tau\simeq\mathbf{s}(\tau')$ or (2) $u=a$ and $\tau=\langle\top,S_0\rangle$. Thus a general term is sendable if it can be derived from a sendable type environment.

The rules for processes also require minor modifications. We can also create a process by spawn if it is sendable. In $\textsc{Out}_d$ we require that values which will be sent across locations have sendable types. However if the transmission is only done in the same location, this condition should be relaxed; in $\textsc{Out}_l$, a message is guaranteed to transmit inside the same location since name a has an input capability. The side condition of LET plays the same role of that of the arrow type in Figure 8. Note also an input process has always the non-sendable type `proc`.

EXAMPLE 4.1. (sq-server) In the following, we offer a non-trivial example of the use of sendability in typing. Recall Examples 2.1 and 2.2, and let us define

$$\sigma=(\mathtt{int},(\mathtt{int})^{\mathtt{0}})^{\mathtt{I}}\qquad\tau=\sigma\to\mathtt{proc}\qquad\sigma'=(\mathtt{int},(\mathtt{int})^{\mathtt{0}})^{\mathtt{IO}}$$

First we note $\lambda(x{:}\sigma).\ \mathbf{sq}(x)$ has a sendable type $\mathbf{s}(\sigma\to\mathtt{proc})$ by Example 2.2. Then **SqServ** is typed as follows.

$$\mathrm{req}:((\tau)^{\mathtt{0}})^{\mathtt{I}}\vdash_{\mathtt{l}} *\mathrm{req}(r:(\tau)^{\mathtt{0}}).r!\langle\lambda(x:\sigma).\ \mathbf{sq}(x)\rangle:\mathtt{proc}$$

Here a type declaration "$(\tau)^{\mathtt{0}}$" 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_I 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 contravariance of output capability:

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

Combining these three, we now infer:

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

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

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

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 [19, 23, 26, 27]. □

One can also check the first system in Example 3.3 and the first and third systems in Example 3.4 are not typable in any environment, while the second systems in Examples 3.3 and 3.4 can be typed using TERM_I and OUT_I rules, respectively (see [33]).

SUBJECT REDUCTION In the following we prove locality is preserved under reduction. Note if P is a function, it is possible that $\Gamma \vdash_1 P : \mathbf{s}(\rho)$ is inferred by either CONST_I,CHAN_I,SUB,APP or LET without using TERM_I directly. But by (1) below, we can regard all sendable types are inferred by TERM_I uniformly. We note that as a special case of (2), we have $\rho = \mathbf{s}(\rho')$.

LEMMA 4.2.

(1) (sendable) $\Gamma \vdash_1 P : \mathbf{s}(\rho)$ *implies there exists* Δ *s.t.* $\Delta \geq \Gamma$, $\Delta \vdash_1 \mathtt{SBL}$, *and* $\Delta \vdash_1 P : \rho$.
(2) (local substitution) *Suppose* $\Gamma, x{:}\tau, \vdash_1 P : \rho$ *and* $\Gamma \vdash_1 V : \tau$. *Then we have* $\Gamma \vdash_1 P\{V/x\} : \rho$.

The following decomposition lemma says the global system is decomposed into a local process with an input-disjoint typed environment.

LEMMA 4.3. *Suppose* $\Gamma \vdash_1 N \equiv (\nu \tilde{a}{:}\tilde{\sigma})((P_{11} \mid \cdots \mid P_{1n_1}) \parallel \cdots \parallel (P_{m1} \mid \cdots \mid P_{mn_m}))$ *where* P_{ij} *is neither* $P \mid Q$ *nor* $(\nu a)P$. *Then there exists* Γ_i *such that* $\tilde{a}{:}\tilde{\sigma}, \Gamma \stackrel{def}{=} \sqcap\Gamma_i$ *with* $\Gamma_i \asymp_1 \Gamma_j$ $(1 \leq i \neq j \leq m)$ *and* $\Gamma_i \vdash_1 P_{ik} : \mathtt{proc}$ $(1 \leq k \leq n_i)$.

The main lemma requires the order-theoretic property, FBC, of our subtyping relation (see [33]), together with Lemma 4.2.

LEMMA 4.4. **(main lemma)** *Suppose* $x{:}\tau', \Gamma_1 \vdash_1 P : \pi$ *and* $\Gamma_2 \vdash_1 V : \mathbf{s}(\tau)$ *with* $\Gamma_1 \asymp_1 \Gamma_2$ *and* $\tau' \leq \mathbf{s}(\tau)$. *Then there exists* Δ *such that* (a) $\Delta \vdash_1 \mathtt{SBL}$, (b) $\Delta \vdash_1 V : \mathbf{s}(\tau)$, (c) $\Gamma_1 \sqcap \Delta \vdash_1 P\{V/x\} : \pi$, (d) $\Gamma_1 \sqcap \Delta \sqcap \Gamma_2 = \Gamma_1 \sqcap \Gamma_2$, and (e) $\Gamma_1 \asymp_1 \Delta \asymp_1 \Gamma_2$.

THEOREM 4.5. **(Subject Reduction Theorem)**

$$\textit{If } \Gamma \vdash_1 N \textit{ and } N \longrightarrow\!\!\!\!\!\rightarrow M, \textit{ then } \Gamma \vdash_1 M.$$

OUTLINE OF THE PROOF: The non-trivial case of the proof of the subject reduction theorem is when a value is sent to a different location by (com_s) rule. By Lemma 4.3, we only consider the following case: suppose $\Gamma_1 \vdash_1 a?(x{:}\tau).\,P$, $\Gamma_2 \vdash_1 a!\langle V\rangle.\,Q$ and $\Gamma_1 \asymp_1 \Gamma_2$. Then we show:

$$\Gamma_1 \sqcap \Gamma_2 \vdash_1 a?(x{:}\tau).\,P \,\|\, a!\langle V\rangle.\,Q \text{ implies } \Gamma_1 \sqcap \Delta \vdash_1 P\{V/x\} \text{ and } \Gamma_2 \vdash_1 Q$$

with $\Gamma_1 \sqcap \Delta \asymp_1 \Gamma_2$ and $\Gamma_1 \sqcap \Delta \sqcap \Gamma_2 = \Gamma_1 \sqcap \Gamma_2$ for some Δ. By the main lemma, we can take Δ as a sendable environment such that $\Delta \vdash_1 V : \mathbf{s}(\tau)$, which establishes the theorem. See [33] for the details.

Combining Theorem 3.2 and Theorem 4.5, we now have:

COROLLARY 4.6. (Type Safety) $\quad \Gamma \vdash_1 N \textit{ and } N \longrightarrow\!\!\!\!\!\rightarrow M \textit{ imply } M \stackrel{\mathit{lerr}}{\not\longrightarrow}.$

5 Further Development

This section illustrates usefulness of our typing system showing some interesting applications.

5.1 Generalisation to Global/Local Subtyping

In this subsection, we show that our sendability notion can be generally extended to a more refined channel usage based on Sewell's idea [29]. A simple extension is easily done by labelling channel types by one of the *locality modes*, given by $\{\mathtt{GG}, \mathtt{LG}, \mathtt{GL}, \mathtt{LL}\}$, ranged over by $m, m', \ldots$, which are:

- $\mathtt{GG}$ – 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 this set is given by: reflexive closure of $\mathtt{GG} \leq m$ with $m = \mathtt{LG}, \mathtt{GL}$ and $m \leq \mathtt{LL}$. Then the syntax of channel type is extended to $m\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$.[2] For the typing system, we first extend the CHAN_l rule in Figure 9 as following CHAN_g rule.

$$\frac{\Gamma \vdash_{\mathtt{g}} a : \mathtt{LL}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle}{\Gamma \vdash_{\mathtt{g}} a : \mathbf{s}(\mathtt{GG}\langle \top, -\rangle)} \quad \frac{\Gamma \vdash_{\mathtt{g}} a : \mathtt{LG}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle}{\Gamma \vdash_{\mathtt{g}} a : \mathbf{s}(\mathtt{GG}\langle \top, S_{\mathtt{O}}\rangle)} \quad \frac{\Gamma \vdash_{\mathtt{g}} a : \mathtt{GL}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle}{\Gamma \vdash_{\mathtt{g}} a : \mathbf{s}(\mathtt{GG}\langle S_{\mathtt{I}}, -\rangle)} \quad \frac{\Gamma \vdash_{\mathtt{g}} a : \mathtt{GG}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle}{\Gamma \vdash_{\mathtt{g}} a : \mathbf{s}(\mathtt{GG}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle)}$$

If a has the mode $\mathtt{GL}$, then it is prohibited from being used as the output subject in an other location, hence it should be sent as if it were only input capability $\langle S_{\mathtt{I}}, -\rangle$ with the mode $\mathtt{GG}$. Since the receiver only accepts the higher capability from the sender, it is only possible to use a as $m\langle S_{\mathtt{I}}, -\rangle$ with $\mathtt{GG} \leq m$. The basic idea of the input and output rules is the same as the rules in Figure 9; we only allow processes to pass sendable terms to a remote location, while any terms can be passed inside its location, whose destination is guaranteed by the mode $\mathtt{LL}$. Thus, we can consider that the local typing system discussed in the previous section is concerned with only sub-ordering $\mathtt{LG} \leq \mathtt{LL}$. In the following, we assume $\mathtt{L}(\tilde{\tau})^{\mathtt{O}}$ denotes either $\mathtt{GL}\langle \top, \tilde{\tau}\rangle$ or $\mathtt{LL}\langle \top, \tilde{\tau}\rangle$, and $\mathtt{L}(\tilde{\tau})^{\mathtt{I}}$ denotes either $\mathtt{LG}\langle \tilde{\tau}, -\rangle$ or $\mathtt{LL}\langle \tilde{\tau}, -\rangle$.

IN_g:
$$\frac{\Gamma \vdash_{\mathtt{g}} u : m(\tilde{\tau})^{\mathtt{I}} \quad \Gamma, \tilde{x}{:}\tilde{\tau} \vdash_{\mathtt{g}} P : \pi \quad (1)}{\Gamma \vdash_{\mathtt{g}} u?(\tilde{x}{:}\tilde{\tau}).P : \pi}$$

OUT_{gd}:
$$\frac{\Gamma \vdash_{\mathtt{g}} u : m(\mathbf{s}(\tilde{\tau}))^{\mathtt{O}} \quad \Gamma \vdash_{\mathtt{g}} V_i : \mathbf{s}(\tau_i) \quad \Gamma \vdash_{\mathtt{g}} P : \pi \quad (1)}{\Gamma \vdash_{\mathtt{g}} u!\langle \tilde{V}\rangle P : \pi}$$

OUT_{gl}:
$$\frac{\Gamma \vdash_{\mathtt{g}} u : \mathtt{LL}\langle (\tilde{\tau}'), (\tilde{\tau})\rangle \quad \Gamma \vdash_{\mathtt{g}} V_i : \tau_i \quad \Gamma \vdash_{\mathtt{g}} P : \mathtt{proc}}{\Gamma \vdash_{\mathtt{g}} u!\langle \tilde{V}\rangle P : \mathtt{proc}}$$

[2] m denotes how a is used as the *subject*, while $S_{\mathtt{I}}, S_{\mathtt{O}}$ in $\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$ stand for types of *objects* a carries; these are notations of the different level. For example, $\mathtt{GL}\langle \tilde{\tau}, \tilde{\tau}'\rangle$ does not mean $\langle \mathtt{G}(\tau), \mathtt{L}(\tilde{\tau}')\rangle$.

where (1) if $m = \mathtt{L}$, then $\pi = \mathtt{proc}$. Note in this generalisation, a term which includes input capabilities can be passed outside if all subjects are global. Other rules except CHAN$_l$, IN and OUT$_{l,g}$ are the same as the rules in Figure 9 replacing $\vdash_1$ with $\vdash_g$.

Then two environments Γ_1 and Γ_2 are composable, denoted by $\Gamma_1 \asymp_g \Gamma_2$, if $\Gamma_1 \asymp \Gamma_2$ and if $u{:}m_i\langle S_{i\mathtt{I}}, S_{i\mathtt{0}}\rangle \in \Gamma_i$ $(i = 1,2)$, then (1) $m_i = \mathtt{LL}$ implies $S_{j\mathtt{I}} = \top$ and $S_{j\mathtt{0}} = -$, (2) $m_i = \mathtt{LG}$ implies $S_{j\mathtt{I}} = \top$, and (3) $m_i = \mathtt{GL}$ implies $S_{j\mathtt{0}} = -$ with $i \neq j$. Then we change the rule of the system composition as in the following.

$$\text{PAR}_g\text{:}\quad \frac{\Gamma \vdash_g M \quad \Delta \vdash_g N \quad \Gamma \asymp_g \Delta}{\Gamma \sqcap \Delta \vdash_g M \,\|\, N}$$

THEOREM 5.1. *(Subject Reduction) If $\Gamma \vdash_g N$ and $N \longrightarrow\!\!\!\!\!\rightarrow M$, then $\Gamma \vdash_g M$.*

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 [23] or [16]. 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

A precise type abstraction of the communication structure of processes induces a non-trivial behaviour equivalence; such effects of types have already been studied in, e.g. [23, 27, 31], for various kinds of encodings. Since we can express computational constraints similar to those in [21, 3, 27] by appropriate restriction of syntax and rules in D$\pi\lambda$, interesting behavioural equalities can be inherited from them for higher-order processes. Let $\approx_\Gamma$ denote a typed barbed reduction-closed congruence defined by input/output predicates and reduction-closure property as in [23, 31, 2, 27]. Then we immediately observe: $\Gamma \vdash N$ and $N \longrightarrow N'$ by either (β), ($\text{let}_{1,2}$) or ($\text{app}_{l,r}$) in Figure 2, then $N \approx_\Gamma N'$. Note also that we do not allow $P \,\|\, (Q \,|\, R)$ being structurally equivalent to $Q \,\|\, (P \,|\, R)$, but we can prove the distributed equations: for some Δ, $(P \,|\, \mathtt{Spawn}(Q)) \,\|\, R \approx_\Delta (P \,\|\, Q) \,\|\, R \approx_\Delta P \,\|\, (Q \,|\, R)$ and $P \,\|\, (a!\langle\tilde{V}\rangle \,|\, Q) \approx_\Delta (P \,|\, a!\langle\tilde{V}\rangle) \,\|\, Q$, etc.

We also have the following multiple higher-order strong replication theorem which is not valid in untyped D$\pi\lambda$, but valid in the local D$\pi\lambda$ studied in Section 4.

PROPOSITION 5.2. *Let us define $R \stackrel{def}{=} *a?(\tilde{x}).\, R_1 \,|\, \cdots \,|\, *a?(\tilde{x}).\, R_n$ with R_i sendable. Then we have:* $\quad (\nu a)(R \,\|\, P \,\|\, Q) \;\approx_\Gamma\; (\nu a)(R \,|\, P) \,\|\, (\nu a)(R \,|\, Q)$

Note we do not require any side condition for P and Q (cf. [23]). The proof is by observing that P and Q may only export the sendable value V via a since the left-hand side of the equation is typable (note it is impossible that a name a is local in either P or Q). We can then apply the standard reasoning framework from [26, 23, 27, 2]. Note also this proposition can not be derived from the system in [27] since a is neither linear/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. [27, 31]) to distributed higher-order processes is an interesting research topic 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, we can construct an equivalent typing system to $\vdash_1$ without TERM$_l$, which becomes syntax-directed (in particular, in type reconstruction we use the partial meet operator to obtain a sendable type). Using this,

we can easily obtain an algorithm to check the typability of P against Γ, as well as an algorithm to compute ρ such that $\Gamma \vdash_1 P : \rho$ given Γ and P. The algorithm can also be extended to the global/local subtyping discussed in 5.1.

6 Discussion and Related Work

We proposed a static local typing system and we used it to show that a a global safety condition can be guaranteed by static type-checking each local configuration; we do not require additional resource informations in the different locations to ensure a safe higher-order process passing/receiving. We also showed that the system can be extended to global/local subtyping [29] and gave some non-trivial equational properties of the underlying languages.

6.1 Encoding into π-calculus

In [26] there is an elegant translation of processes using higher order values into a first order process language where only channel names are transmitted. We now examine this translation of D$\pi\lambda$. The basic idea 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

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

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 already has a square server for faster parallel evaluation.

$$\mathbf{Client}_2 \Leftarrow\!= (\nu\, ar)(\mathrm{req}!\langle r\rangle.\, r?(X).\, X\, a \mid \mathbf{sq}(a) \mid a!\langle 1, c_1\rangle \mid a!\langle 2, c_2\rangle \mid a!\langle 3, c_3\rangle \cdots)$$

Then the client is replaced by

$$[\![\mathbf{Client}_2]\!] \Leftarrow\!= (\nu\, ar)(\mathrm{req}!\langle r\rangle.\, r?(\mathrm{tr}).\, \mathrm{tr}!\langle a\rangle \mid \mathbf{sq}(a) \mid a!\langle 1, c_1\rangle \mid a!\langle 2, c_2\rangle \mid a!\langle 3, c_3\rangle \cdots)$$

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 differences between the two systems:

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

As seen in the following, 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, which disturbs the locality.

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

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

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 4.1. This example shows that it would be extremely difficult to adapt the translation technique in [26] so that the local typing structure is preserved. This indicates higher order distributed calculi are worthy of independent investigation.

6.2 Related work

Locality in π-calculus: The expressiveness of locality and mobility have been studied in the π-calculus, especially in [3, 21, 5, 32, 11]. The untyped local π-calculus [21, 5, 32] is simply defined with the following input restriction rule

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

By Corollary 4.6, the proper subset of $D\pi\lambda$ in Section 4 where we only consider channel passing and a single location without using OUT_l rule, satisfies their required locality property. Similarly *receptiveness* studied in [3, 27] is ensured if the system starts reductions from the initial statement where each location contains only a single process.[3] One may consider a better way to ensure the locality in $D\pi\lambda$ is the following syntactic restriction as the same as the local π-calculus.

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

However this restriction is too strong; if we use it, new receptors are never created by β-reduction, hence Example 4.1 is no longer typable. Moreover this idea does not work if we wish to control the higher-order abstraction as seen in Example 3.4.

[11] showed that a locality condition similar to ours of Section 4 is practically useful to describe various kinds of encodings in Distributed Join-Calculus. Our approach differs from theirs since our aim is to establish a formal typing system for arbitrary higher-order process passing and instantiations which ensures locality in the non-local environments; our typing system control higher-order term passing where new receptors can be created inside the same location (cf. OUT_l in Figure 9).

Restriction of capability on names: In our system, a channel which has an input capability in the local environment can be exported outside the original location *as if it had output capability only*, by CHAN_l in Figure 9; this is essential to represent a continuation passing style programming as seen in Example 4.1, which is in some aspects similar to a formulation of triggers in Definition 5.3.1 in [23]. Secondly in our system, if a variable appears in an object position in an outgoing message, it should have a sendable type. Hence if it is bound by a prefix or λ-abstraction, then the bound variable is automatically annotated by $\mathbf{s}(\ )$ (e.g. $b?(Y:\mathbf{s}(\tau)).\, d!\langle Y\rangle$), which rejects receipt of non-sendable value from a different location. The idea of direct restrictions on input prefixes rather than on outgoing values is also studied in [4] for the π-calculus, with the aim of ensuring no leak of secret names. In our system these frameworks are generalised to deal with higher-order process passing preserving locality with subtyping of λ and π-calculi.

Distributed higher-order processes: As argued in [10, 9, 20, 28], many practical applications call for parameterised higher-order process passing, which may be difficult to represent directly without functional constructions even with a migration of the second order processes. Since our language is also based on the subtyping of the λ-calculus, it is straightforward to import richer subtyping, e.g. records, recursive types, polymorphic types into the distributed languages.

We believe our simple capability control based on subtyping will be equally applicable to a wide range of concurrent/functional languages, including [6, 10, 9, 20, 28]. In general, extensions of higher-order process passing to more advanced distributed primitives, such as hierarchical location spaces [16, 29, 15], process mobility [7, 28, 11], and cryptographic constructs [1, 14] needs to be investigated.

References

[1] Abadi, M. and Gordon, A., The Spi-calculus, Computer and Communications Security, 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] C. Bodei et al., Control Flow Analysis for the π-calculus, *Proc. CONCUR'98*, pp.85–98, LNCS 1466, Springer-Verlag, 1998.

[3] More exactly, we need to replace $*P$ with the input replication $*a?(x).\, P$ and use the demand driven reduction for $*a?(x).\, P$ as (com_s) instead of $*P \equiv P \mid *P$. For the linear and ω receptiveness, we need an additional constraint such that u is not local under Γ in IN in Figure 9. Note we do not treat uniformity [27]/dead-lock freedom [31] for the simplicity.

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

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

[7] Cardelli, L. and Gordon, A., Typed Mobile Ambients, *Proc. POPL'99*, ACM Press, 1999.

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

[9] Ferreira, W., Hennessy, M. and Jeffrey, M., A Theory of Weak Bisimulation for Core CML, Proc. Int. Conf. Functional Programming, pp.201–212, ACM Press, 1996.

[10] 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.

[11] Fournet, C. et al., 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., Type Safe Extension of Mobile Agents in Anonymous Networks, *POPL'99*, ACM Press, 1999.

[16] 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.

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

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

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

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

[21] Merro, M. and Sangiorgi, D., On asynchrony in name-passing calculi, *ICALP'98*, 1998.

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

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

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

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

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

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

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

[29] Sewell, P., Global/Local Subtyping for a Distributed pi-calculus. Technical Report 435, Computer Laboratory, University of Cambridge, 1997. Extended Abstract appeared in *Proc. ICALP'98*.

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

[31] 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.

[32] 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.

[33] Full version of this paper. Available from: http://www.cogs.susx.ac.uk/users/nobuko/index.html. To appear as CS Report, University of Sussex, 1999.