Subtyping and Locality in Distributed Higher Order Processes[†]
(extended abstract)

Nobuko Yoshida and Matthew Hennessy

COGS, University of Sussex

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, 10, 20, 25, 30]. Most of these are based on first-order extensions of the π-calculus [21]; 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 [27]. 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 integration of the call-by-value λ-calculus and the π-calculus [21], 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, 9, 19], CML [8] and LLinda [22], and can be regarded as an extension of Blue-calculus [5] to a higher-order term passing.

A desirable feature of some distributed systems is that every channel name is associated with a *unique* receptor, which is called *receptiveness* in [28]; another property called *locality* where new receptors are not created by received channels, has also been studied in [3, 4, 20, 34] for an asynchronous version of the π-calculus [16]. The combination of these constraints provides a model of a realistic distributed environment, which

---

[†] Supported by EPSRC GR/K60701 and CONFER II. E-mail:{`nobuko,matthewh`}`@cogs.susx.ac.uk`.

$$\begin{array}{lrl}
\text{Term:} & P,Q,\ldots ::= & V \mid PQ \mid \\
 & & u?(\tilde{x}:\tilde{\tau}).P \mid u!\langle\tilde{V}\rangle P \mid (\nu a:\sigma)P \mid *P \mid P|Q \mid \mathbf{0} \\
\text{Value:} & V,W,\ldots ::= & u \mid \lambda(x:\tau).P \\
\text{Identifier:} & u,v,\ldots ::= & l \mid a \mid x \\
\text{Literal:} & l,l',\ldots ::= & \texttt{true} \mid \texttt{false} \mid () \mid 0 \mid 1 \mid \ldots
\end{array}$$

Figure 1. Syntax of $\pi\lambda$

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 [10]; 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 [7]; This locality constraint should be obeyed even in the presence of parameterised object passing, which is recently often found in practice [11].

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 [14, 24] to a higher-order setting where the order theoretic property of subtyping relation, *finite-bounded completeness*, 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: Section 2 studies a call-by-value higher-order π-calculus, $\pi\lambda$, with subtyping. Section 3 introduces a distributed version of $\pi\lambda$, called D$\pi\lambda$, and illustrates the difficulty of enforcing locality in D$\pi\lambda$. Section 4 proposes a new typing system to ensure the locality. Section 5 discusses applications of our type discipline; extendibility of our system to more general global/local channel constraints studied in [30] in a higher-order setting, a multiple higher-order replication theorem extending [24], and the type checking. Section 6 concludes with discussions and related work. Due to space limitation, we leave the detailed explanations and proofs to the full version [35].

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

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. The syntax is a mixture of

Reduction Rules:

$$(\beta) \quad (\lambda(x\!:\!\tau).P)V \longrightarrow P\{V/x\} \qquad (\mathrm{app}_l)\ \frac{P \longrightarrow P'}{PV \longrightarrow P'V}$$

$$(\mathrm{com}) \quad u?(\tilde{x}\!:\!\tilde{\tau}).P \,|\, u!\langle\tilde{V}\rangle Q \longrightarrow P\{\tilde{V}/\tilde{x}\} \,|\, Q \qquad (\mathrm{app}_r)\ \frac{Q \longrightarrow Q'}{PQ \longrightarrow PQ'}$$

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

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 $a \notin \mathsf{fn}(Q)$

Figure 2. Reduction for $\pi\lambda$

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. From the latter we have input and output on communication channels, dynamic channel creation, iteration and the empty process. We use the standard notational conventions; for example ignoring trailing occurrences of $\mathbf{0}$ and omitting type annotations unless they are relevant. We use $\mathsf{fn}(P)$/$\mathsf{fv}(P)$ to denote the sets of *free names/variables* respectively, and typically write $\lambda().P$ for a *thunk* of P, $\lambda(x\!:\!\texttt{unit}).P$ assuming $x \notin \mathsf{fv}(P)$.

REDUCTION The reduction semantics of $\pi\lambda$ is given in Figure 2. The main reduction rules are β-reduction, (β), and communication, (com). The final contextual rule, (str), uses a structural rules from the π-calculus. We use $\longrightarrow\!\!\!\!\rightarrow$ to denote multi-step reductions.

EXAMPLE 2.1. (sq-server) Suppose that in the language we have a literal `sq` for squaring natural numbers. 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 squaring data and then returns it along the return channel.

A sq-server, **sqServ**, is a process which on requests sends to the client the code for squaring values, which the client can initialise locally.

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

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 a local channel a which will act as the request channel for data processing:

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

Type:

Term Type: $\rho \ ::= \ \mathtt{proc} \mid \tau$

Value Type: $\tau \ ::= \ \mathtt{unit} \mid \mathtt{bool} \mid \mathtt{nat} \mid \tau \to \rho \mid \sigma$

Channel Type: $\sigma \ ::= \ \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$ 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) $\sigma_i = \langle S_{i\mathtt{I}}, S_{i\mathtt{O}} \rangle,\ S_{1\mathtt{I}} \leq S_{2\mathtt{I}},\ S_{1\mathtt{O}} \geq S_{2\mathtt{O}} \ \Rightarrow \ \sigma_1 \leq \sigma_2.$

Abbreviations:

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

Figure 3. Types for $\pi\lambda$

IO-TYPES We use as types for $\pi\lambda$ a simplification of the input/output capabilities of [14] (in turn a *strict* generalisation of [24][1]). 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 may then be constructed from these types using the constructor $\to$, as in the λ-calculus. 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 [14, 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 always takes fewer capabilities than specified by the outside environment, while a sender always send more capabilities than specified. Then IO-types in [24] are represented as a special case of our IO-types; to denote them, we introduce the abbreviations in Figure 3. Note that $(\tilde{\tau})^{\mathtt{IO}} \leq (\tilde{\tau})^{\mathtt{I}} \leq \langle \top, \bot \rangle$ and $(\tilde{\tau})^{\mathtt{IO}} \leq (\tilde{\tau})^{\mathtt{O}} \leq \langle \top, \bot \rangle$.[2]

The subtyping relation over types defined in Figure 3 is partial order and *finite bounded complete*, FBC, (cf. [14]). The partial meet operator $\sqcap$ and partial join operator $\sqcup$ can be also defined directly following [14]. For the base and arrow types, we define $\sqcap/\sqcup$ as the standard meet/join operators w.r.t. $\leq$. For channel types, we use the following definition:

$$\text{(vec)} \quad (\tilde{\tau}) \sqcup (\tilde{\tau}') \overset{\text{def}}{=} (\tilde{\tau}'') \text{ with } \tau''_i = \tau_i \sqcup \tau'_i \quad \text{and} \quad (\tilde{\tau}) \sqcap (\tilde{\tau}') \overset{\text{def}}{=} (\tilde{\tau}'') \text{ with } \tau''_i = \tau_i \sqcap \tau'_i$$

---

[1] Our general form of IO-types gives more typable terms than [24] even if we restrict our language to the pure polyadic π-calculus. See Example 2.5 in [35].

[2] 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. The side coditions $S_{\mathtt{I}} \neq \bot$ and $S_{\mathtt{O}} \neq \top$ ensure to avoid mismatching arity constraints.

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:

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

Process Typing Rules:

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

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

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

Figure 4. Typing System for πλ

(chan) $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \sqcup \langle S'_{\mathtt{I}}, S'_{\mathtt{O}} \rangle \overset{\text{def}}{=} \langle S_{\mathtt{I}} \sqcup S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcap S'_{\mathtt{O}} \rangle$ and
$\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \sqcap \langle S'_{\mathtt{I}}, S'_{\mathtt{O}} \rangle \overset{\text{def}}{=} \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}} \rangle$ if $S_{\mathtt{I}} \geq S'_{\mathtt{O}}$ and $S'_{\mathtt{I}} \geq S_{\mathtt{O}}$; else undefined.

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, 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:

(1) $\mathsf{dom}(\Gamma)$ denotes $\{u \mid u{:}\tau \in \Gamma\}$ and Γ/A denotes $\{u{:}\tau \in \Gamma \mid u \notin A\}$.
(2) $\Gamma, u{:}\tau$ means $\Gamma \cup \{u{:}\tau\}$, together with the assumption $u \notin \mathsf{dom}(\Gamma)$.
(3) $\Delta \leq \Gamma$ means $\Delta(u) \leq \Gamma(u)$ for all $u \in \mathsf{dom}(\Gamma)$.

Then we define the partial meet operator $\sqcap$ and the partial join operator $\sqcup$ as:

$$\Gamma \sqcap \Delta \overset{\text{def}}{=} \Gamma/\mathsf{dom}(\Delta) \cup \Delta/\mathsf{dom}(\Gamma) \cup \{u{:}(\Delta(u) \sqcap \Gamma(u)) \mid u \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)\} \text{ and}$$
$$\Gamma \sqcup \Delta \overset{\text{def}}{=} \{u{:}(\Delta(u) \sqcup \Gamma(u)) \mid u \in \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta)\}$$

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 [14, 24].

EXAMPLE 2.2. (typed sq server) We now revisit Example 2.1. 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 we type it as:

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

Syntax:

System: $M, N, \ldots \quad ::= \quad P \;\mid\; N \,||\, M \;\mid\; (\nu a{:}\sigma)N$
Term: $P, Q, \ldots \quad ::= \quad \mathtt{Spawn}(P) \;\mid\; \cdots$ from Figure 1
Value: as in Figure 1

Distributed Reduction Rules:

$$\text{(spawn)} \quad (\cdots Q \,|\, \mathtt{Spawn}(P)) \longrightarrow (\cdots Q) \,||\, P$$

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

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

Figure 5. Syntax and Distributed Reduction in Dπλ

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

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

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. □

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

THEOREM 2.3. (Subject Reduction) *If* $\Gamma \vdash P : \rho$ *and* $P \longrightarrow\!\!\!\!\to P'$, *then* $\Gamma \vdash P' : \rho$.

3 Locality of Channels in Distributed Higher Order π-calculus

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. The reduction semantics of the previous section is extended to the new language, Dπλ, 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:

Input Predicate:

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

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

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

Local Distributed Rules:

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

INTRO: $$\frac{\Gamma \vdash P : \mathtt{proc}}{\Gamma \vdash_{\mathtt{l}} P}$$

PAR$_l$: $$\frac{\Gamma \vdash_{\mathtt{l}} N \quad \Delta \vdash_{\mathtt{l}} M \quad \Gamma \asymp_{\mathtt{l}} \Delta}{\Gamma \sqcap \Delta \vdash_{\mathtt{l}} N \,||\, M}$$

RES$_l$: $$\frac{\Gamma, a{:}\sigma \vdash_{\mathtt{l}} M}{\Gamma \vdash_{\mathtt{l}} (\nu a{:}\sigma)M}$$

Figure 7. Local Distributed Typing Rules

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 uses an input predicate $P \downarrow a^{\mathtt{I}}$ 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 $\sigma = \langle(\tilde{\tau}), S_{\mathtt{O}}\rangle$. We also call u is *local under* Γ if $\Gamma(u)$ is local.

DEFINITION 3.1. (system composable) 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 $\Gamma_1 \asymp_{\mathtt{l}} \Gamma_2$, 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 $S_{2\mathtt{I}} = \top$. □

Intuitively this means that if a channel a is local in Γ_1, then it must not be local in another environment Γ_2.

The typing system for distributed systems given in Figure 7 is simply in form of $\Gamma \vdash_{\mathtt{l}} N$ where Γ is again the same environment. The most essential rule is PAR$_l$; this says that $N_1 \,||\, N_2$ is typable with respect to Δ if Δ can be written as $\Gamma_1 \sqcap \Gamma_2$, where $\Gamma_1 \asymp_{\mathtt{l}} \Gamma_2$ and N_i is typable with respect to Γ_i. If terms are system composable, then we have no immediate locality error since $P \downarrow a^{\mathtt{I}}$ and $\Gamma \vdash_{\mathtt{l}} P : \mathtt{proc}$ imply $\Gamma \vdash a : (\tilde{\tau})^{\mathtt{I}}$ for some $\tilde{\tau}$. That is:

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

It is however easy to see that system composability as defined above is not closed under reduction: indeed, we easily have $N \not\xrightarrow{lerr}$ and $N \twoheadrightarrow N'$ 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 is the use of a name received from another location as an input subject. Take $a?(x).\, P \,|\, b!\langle a\rangle \parallel b?(y).\, y?(z).\, Q$. Then it is easy to check that this can be typed with $\textsc{Par}_l$ in Figure 7. However after one reduction step, the communication along b, we obtain $a?(x).\, P \parallel a?(z).\, Q$, which is no longer typable. It is not difficult to exclude such terms which do not involve term passing by a simple syntactic condition or typing systems as studied in [2, 4, 20, 28]. The second, which is more complicated, concerns the parameterisations of processes and the instantiation of variables which occur in outgoing values. The presence of higher-order passing makes the problem subtle, as seen in the next example.

EXAMPLE 3.3. Let V denote the value $\lambda().\,\mathbf{sq}(a)$ in the slightly modified system

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

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 $\lambda(x).\,\mathbf{sq}(x)$.

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

This does not destroy locality while it creates a new receptor $\mathbf{sq}(c)$. However the following system which sends the same value as the above to b disturbs locality.

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

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; in the third system, V is transmitted along b across locations, where it is used to dynamically construct a new dangerous value $\lambda().(V\, c)$; this is then transmitted across locations via d and when it is run we obtain once more a locality error.[3] We need a new set of *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

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 [23].

The first extra ordering ensures that $\leq$ is a preorder. The second 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 $\simeq \overset{\text{def}}{=} \leq \cap \geq$. Similarly with (lift), we have: $\mathbf{s}(\mathbf{s}(\tau) \to \mathbf{s}(\rho)) \simeq \mathbf{s}(\tau) \to \mathbf{s}(\rho)$. Based on this ordering, we formalise *sendable types* as follows.

---

[3] See [35] for further non-trivial examples of higher-order processes.

Types:

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

Ordering: All rules from Figure 3 and

$$
\begin{array}{llll}
\text{(trans)} & \rho_1 \leq \rho_2 \quad \rho_2 \leq \rho_3 \;\Rightarrow\; \rho_1 \leq \rho_3 & \text{(id)} & \mathbf{s}(\rho) \leq \mathbf{s}(\mathbf{s}(\rho)) \\
\text{(mono)} & \rho \leq \rho' \;\Rightarrow\; \mathbf{s}(\rho) \leq \mathbf{s}(\rho') & \text{(lift)} & \mathbf{s}(\tau) \rightarrow \mathbf{s}(\rho) \leq \mathbf{s}(\mathbf{s}(\tau) \rightarrow \mathbf{s}(\rho)) \\
\text{(sendable)} & \mathbf{s}(\rho) \leq \rho & &
\end{array}
$$

Figure 8. Locality types for Dπλ

DEFINITION 4.1. Let Sble, the set of sendable types, be the least set of types which includes all types of the form $\mathbf{s}(\rho)$, and for which τ, $\rho \in \mathsf{Sble}$ implies $\tau \rightarrow \rho \in \mathsf{Sble}$. 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 4.2. (downwards closed) (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* $\rho \simeq \mathbf{s}(\rho)$, *and* (3) $\rho \in \mathsf{Sble}$ *if and only if* $\rho \leq \mathbf{s}(\rho')$ *for some* ρ'.

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.

The essential order theoretic property, FBC, is also preserved on this subtyping relation. We extend the definition of $\sqcap$ and $\sqcup$ in § 2.3 to the sendable types as follows.

$$
\begin{array}{lll}
\mathbf{s}(\rho_1) \sqcap \mathbf{s}(\rho_2) \;=\; \mathbf{s}(\rho_1 \sqcap \rho_2) & \text{and} & \mathbf{s}(\rho_1) \sqcup \mathbf{s}(\rho_2) \;=\; \mathbf{s}(\rho_1 \sqcup \rho_2) \\
\mathbf{s}(\rho_1) \sqcap \rho_2 \;=\; \mathbf{s}(\rho_1 \sqcap \rho_2) & \text{and} & \mathbf{s}(\rho_1) \sqcup \rho_2 \;=\; \rho_1 \sqcup \rho_2 \quad \text{with } \rho_2 \neq \mathbf{s}(\rho_2').
\end{array}
$$

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*, which only use sendable types.

DEFINITION 4.3. A typing environment Δ is *sendable*, written $\Delta \vdash_1$ SBL, if $u : \tau \in \Delta$ implies $\tau \in \mathsf{Sble}$ or $a : \sigma \in \Delta$ implies $\sigma \in \langle \top, S_{\mathtt{O}} \rangle$. □

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 TERM$_l$, which says that in a general term is sendable only if it can be derived from a sendable type environment. For processes, first we can create a process by spawn only if it is sendable. In 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 should be relaxed; in OUT$_l$, the message is guaranteed to be transmitted within the same location since name a has an

Send Rules:

$$\textsc{Const}_l : \frac{\Gamma \vdash_{\mathtt{1}} l : \mathtt{nat}}{\Gamma \vdash_{\mathtt{1}} l : \mathbf{s}(\mathtt{nat})} \quad \text{etc.} \qquad \textsc{Chan}_l : \frac{\Gamma \vdash_{\mathtt{1}} a : \langle \top, S \rangle}{\Gamma \vdash_{\mathtt{1}} a : \mathbf{s}(\langle \top, S \rangle)}$$

$$\textsc{Term}_l : \frac{\Delta \vdash_{\mathtt{1}} P : \rho \quad \Delta \vdash_{\mathtt{1}} \mathtt{SBL} \quad \Delta \geq \Gamma}{\Gamma \vdash_{\mathtt{1}} P : \mathbf{s}(\rho)} \qquad \textsc{Spawn}_l : \frac{\Gamma \vdash_{\mathtt{1}} P : \mathbf{s}(\mathtt{proc})}{\Gamma \vdash_{\mathtt{1}} \mathtt{Spawn}(P) : \mathbf{s}(\mathtt{proc})}$$

Common Rules: as in Figure 4.

Functional Rules: as in Figure 4.

Process Rules:

$$\textsc{Out}_d : \frac{\begin{array}{l}\Gamma \vdash_{\mathtt{1}} u : (\mathbf{s}\widetilde{(\tau)})^{\mathtt{O}} \\ \Gamma \vdash_{\mathtt{1}} V_i : \mathbf{s}(\tau_i) \quad \Gamma \vdash_{\mathtt{1}} P : \pi\end{array}}{\Gamma \vdash_{\mathtt{1}} u!\langle \tilde{V} \rangle P : \pi} \qquad \textsc{Out}_l : \frac{\begin{array}{l}\Gamma \vdash_{\mathtt{1}} u : \langle (\tilde{\tau}'), (\tilde{\tau}) \rangle \\ \Gamma \vdash_{\mathtt{1}} V_i : \tau_i \quad \Gamma \vdash_{\mathtt{1}} P : \mathtt{proc}\end{array}}{\Gamma \vdash_{\mathtt{1}} u!\langle \tilde{V} \rangle P : \mathtt{proc}}$$

NIL,REP,PAR,RES as in Figure 4 with `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 $\vdash_{\mathtt{1}}$ in INTRO.

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

input capability. Note also an input process has always the non-sendable type `proc`.

EXAMPLE 4.4. (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

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

First we note $\lambda(x{:}\sigma).\, \mathbf{sq}(x)$ has a sendable type $\mathbf{s}(\sigma \rightarrow \mathtt{proc})$; the derivation is similar to that in Example 2.2, followed by an application of $\textsc{Term}_l$. Then **SqServ** is typed as:

$$\mathrm{req} : ((\tau)^{\mathtt{O}})^{\mathtt{I}} \vdash_{\mathtt{1}} *\mathrm{req}?(r : (\tau)^{\mathtt{O}}).r!\langle \lambda(x{:}\sigma).\, \mathbf{sq}(x) \rangle : \mathtt{proc}$$

Next for **Client**, first let us define its body as $P \equiv (\ X\, a\ \ | a!\langle 1, c_1 \rangle \mid \cdots)$. 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: $X : \tau,\ a{:}\sigma' \vdash_{\mathtt{1}} P : \mathtt{proc}$. Now define $\Gamma = \mathrm{req} : ((\tau)^{\mathtt{O}})^{\mathtt{O}}, r : (\tau)^{\mathtt{IO}}$. Then by applying RES and IN, we have:

$$\Gamma \vdash_{\mathtt{1}} r?(X : \tau).\, (\nu a{:}\sigma')P : \mathtt{proc}$$

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

$$\textsc{Chan}_l \quad \frac{\Gamma \vdash_{\mathtt{1}} r : (\tau)^{\mathtt{O}}}{\Gamma \vdash_{\mathtt{1}} r : \mathbf{s}((\tau)^{\mathtt{O}})}$$

The type of the channel "req" in the client is inferred as $\Gamma \vdash_{\mathtt{1}} \mathrm{req} : (\mathbf{s}((\tau)^{\mathtt{O}}))^{\mathtt{O}}$ by $\mathbf{s}((\tau)^{\mathtt{O}}) \leq (\tau)^{\mathtt{O}}$ as well as the contravariance of output capability. Combining these three, we infer:

$$\mathrm{req} : ((\tau)^{\mathtt{O}})^{\mathtt{O}} \vdash_{\mathtt{1}} (\nu r : (\tau)^{\mathtt{IO}})\mathrm{req}!\langle r \rangle r?(X : \tau).\, (\nu a{:}\sigma')P \equiv \textbf{Client} : \mathtt{proc}$$

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

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

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 [16, 24, 27, 28]. Moreover it demonstrates the need for non-trivial subtyping on channels. □

One can refer to [35] for more examples which show that our systems eliminates various forms of behaviour which destroy the locality, like those in Example 3.3.

SUBJECT REDUCTION We now prove locality is preserved under reduction.

LEMMA 4.5. (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) 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) (substitution) *Suppose* $\Gamma, x:\tau \vdash_1 P : \rho$ *and* $\Gamma \vdash_1 V : \tau$. *Then* $\Gamma \vdash_1 P\{V/x\} : \rho$.

The second property, which is needed to prove (3), is the most important. 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$_l$.[4]

The main lemma requires the order-theoretic property, FBC, of our subtyping relation, together with Lemma 4.5.

LEMMA 4.6. **(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 $\Gamma_1 \sqcap \Delta \asymp_1 \Gamma_2$.

The non-trivial case of the proof of the subject reduction property is when a value is sent to a different location by (com_s) rule: 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 must 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 $\Delta \vdash_1 V : \mathbf{s}(\tau)$. Now we establish:

THEOREM 4.7. **(Subject Reduction Theorem)**

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

---

[4] The proof of the second property relies directly on the constraint on the construction of arrow types. Relaxing this constraint would allow us to type 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 research (cf. [23]).

See [35] for the proofs. Combining Theorem 3.2 and Theorem 4.7, we now have:

COROLLARY 4.8. (Type Safety) *$\Gamma \vdash_1 N$ and $N \longrightarrow\!\!\!\!\rightarrow M$ imply $M \overset{lerr}{\not\longrightarrow}$.*

5 Further Development

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 [30], 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 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:

(1) $\mathtt{GG}$ – a channel is allowed to be used as the input and output subjects anywhere.
(2) $\mathtt{GL}$ (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.
(3) $\mathtt{LL}$ – a channel is used as the input and output subjects only in this location.

A partial order on this set is given by a 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 $m\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$ where m denotes how the channel is used as the subject while $S_{\mathtt{I}}, S_{\mathtt{O}}$ stand for the types of objects which it carries.

In the revised system judgements take the form: $\Gamma \vdash_{\mathrm{g}} P : \rho$. First we replace CHAN$_l$ in Figure 9 with a more general rule which indicates when channel capabilities may be transferred between locations:

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

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 $S_{\mathtt{I}}, S_{\mathtt{O}}$. 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:

$$\text{IN}_{gl}: \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{g}} u : \mathtt{LL}(\tilde{\tau})^{\mathtt{I}} \\ \Gamma, \tilde{x}{:}\tilde{\tau} \vdash_{\mathrm{g}} P : \mathtt{proc}\end{array}}{\Gamma \vdash_{\mathrm{g}} u?(\tilde{x}{:}\tilde{\tau}).P : \mathtt{proc}} \quad \text{OUT}_{gl}: \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{g}} u : \mathtt{LL}\langle(\tilde{\tau}'), (\tilde{\tau})\rangle \\ \Gamma \vdash_{\mathrm{g}} V_i : \tau_i \\ \Gamma \vdash_{\mathrm{g}} P : \mathtt{proc}\end{array}}{\Gamma \vdash_{\mathrm{g}} u!\langle\tilde{V}\rangle P : \mathtt{proc}} \quad \text{OUT}_{gd}: \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{g}} u : \mathtt{LL}(\widetilde{\mathbf{s}(\tau)})^{\mathtt{O}} \\ \Gamma \vdash_{\mathrm{g}} V_i : \mathbf{s}(\tau_i) \\ \Gamma \vdash_{\mathrm{g}} P : \mathtt{proc}\end{array}}{\Gamma \vdash_{\mathrm{g}} u!\langle\tilde{V}\rangle P : \mathtt{proc}}$$

Finally to compose systems we need a more general definition of *system composable*: 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:

$$\text{PAR}_g: \quad \frac{\Gamma \vdash_{\mathrm{g}} M \quad \Delta \vdash_{\mathrm{g}} N \quad \Gamma \asymp_{\mathrm{g}} \Delta}{\Gamma \sqcap \Delta \vdash_{\mathrm{g}} M \| N}$$

THEOREM 5.1. (Subject Reduction) *If $\Gamma \vdash_{\mathrm{g}} N$ and $N \longrightarrow\!\!\!\!\rightarrow M$, then $\Gamma \vdash_{\mathrm{g}} M$.*

Type Safety Theorem could easily be established for this typing system, by introducing a *tagged* version of the language (cf. [14, 24, 30]).

5.2 Behavioral Equivalence

Typing systems impose constraints on the communication structure of processes and various authors, for example [24, 28, 33] have used this to define relativised behavioural equivalences. These have proved useful for example in studying the properties of translations between languages [3, 24, 33]. This technique can also be applied to Dπλ, 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 [2, 17, 24, 28, 33]. Various properties of $\approx_\Gamma$ and $\sim_\Gamma$, proved for variations of the π-calculus, can easily be generalised to Dπλ; a simple example is closure under β-reduction (see page 10 in [2]). We can also prove various distributed equations, such as: $(P \mid \mathtt{Spawn}(Q)) \,||\, R \approx_\Delta (P \,||\, Q) \,||\, R \sim_\Delta P \,||\, (Q \mid R)$ and $P \,||\, (a!\langle\tilde{V}\rangle \mid Q) \sim_\Delta (P \mid a!\langle\tilde{V}\rangle) \,||\, Q$.

We also have the following multiple higher-order strong replication theorem which is not valid in Dπλ without types, but valid in the local Dπλ studied in Section 4.

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

Note we do not require any side condition for P and Q (cf. [24]). The proof is by observing that P and Q may only export the sendable value V via a since it is impossible that the name a is local in either P or Q. We can then apply the standard reasoning framework from [24, 27, 28]. See [35] for proofs. Note also that this proposition can not be derived in the framework of [28] 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. [28, 33]) 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, to obtain a syntax directed system (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 $\Gamma \vdash_1 P : \rho$ along the line of [31]. Once processes in each location are type-checked, the algorithm which computes typability of a global system $\Gamma \vdash_1 N$ is simply obtained by decidability of $\Gamma \asymp_1 \Delta$. See [35] for details.

6 Discussion and Related Work

We have proposed a local subtyping system for a simple higher-order distributed process language Dπλ 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 [23], agent migration [26], an implementation of network protocols [18], and a location-based Linda language [22].

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 as in [3, 14], or where there is more significant interplay between the concurrent and the functional language primitives, as in Facile [9, 19]. However extensions of our capability based typing systems to more advanced distributed primitives, such as hierarchical location spaces [32], process mobility [6, 10, 29], and cryptographic constructs [1, 13] will be more challenging. Since in our language we inherit the standard subtyping of the λ-calculus, it is also 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.

It has been argued that in some sense there is no need for higher-order constructs in π-calculus based languages. For example in [27] there is a concise translation of processes using higher order values into the π-calculus. However, as we will now examine in the context of D$\pi\lambda$, certain information is lost in such translations.

The basic idea of the translation in [27] 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

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

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

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

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 exists an essential difference between $\mathbf{sqServ} \parallel \mathbf{Client}_2$ and its translation $\llbracket \mathbf{sqServ} \rrbracket \parallel \llbracket \mathbf{Client}_2 \rrbracket$. 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:

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

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

This disturbs the locality 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 4.4. This example shows that it would be difficult to adapt the translation technique in [27] so that the local typing structure is preserved; it provides at least one reason why higher-order distributed calculi are worthy of investigation independently. Moreover, as argued in [8, 9, 19, 22, 29], 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; their presence leads to a natural and powerful programming style as seen in the above literature.

Preserving the locality of channels has been studied extensively for the π-calculus, [3, 4, 20, 34, 10]. For example the (untyped) *local π-calculus* [4, 20, 34] 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 Dπλ, 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 4.4 would no longer be typable. Moreover this idea does not work if we wish to control higher-order variable, as seen in Example 3.3 (see also [35]).

A locality condition similar to ours is used in [10] 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: We thank James Riely, Kohei Honda and the members of Wednesday Study Group of Sussex University for comments and discussions, and anonymous referees for useful comments.

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, *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, *POPL'99*, pp.79–92, ACM Press, 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, *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, *ICALP'90*, LNCS 443, pp.765–780, Springer-Verlag, 1990.

10. 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.
11. Sun Microsystems Inc., Java home page. http://www.javasoft.com/, 1995.
12. 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.
13. Heintze, N. and Riecke, J., The SLam Calculus: Programming with Secrecy and Integrity, *POPL'98*, pp.365-377. ACM Press, 1998.
14. 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.
15. Honda, K., Composing Processes, *POPL'96*, pp.344-357, ACM Press, 1996.
16. Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. *ECOOP'91*, LNCS 512, pp.133–147, Springer-Verlag 1991.
17. Honda, K. and Yoshida, N., On Reduction-Based Process Semantics. *TCS*, pp.437–486, No.151, North-Holland, 1995.
18. Jeffrey, A. and Wakeman, I., SafetyNet. Available from: http://klee.cs.depaul.edu/an/, 1998.
19. Leth, L. and Thomsen, B., Some Facile Chemistry, ERCC Technical Report, ERCC-92-14, 1992.
20. Merro, M. and Sangiorgi, D., On asynchrony in name-passing calculi, *ICALP'98*, LNCS 1443, pp.856–867, Springer-Verlag, 1998.
21. Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes. *Information and Computation*, 100(1), pp.1–77, 1992.
22. 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), 1998.
23. O'Hearn, P., Power, J., Takeyama, M., and Tennent, D., Syntactic Control of Interference Revised, *MFPS'97*, *ENCS*, Elsevier, 1997.
24. Pierce, B.C. and Sangiorgi. D, Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.
25. Pierce, B. and Turner, D., Pict: A Programming Language Based on the Pi-calculus, Indiana University, CSCI Technical Report, 476, March, 1997.
26. Riely, J. and Hennessy, M., Trust and Partial Typing in Open Systems of Mobile Agents, CS Technical Report, University of Sussex, 04/98, Available at: http://www.cogs.susx.ac.uk, 1998.
27. Sangiorgi, D., *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms*. Ph.D. Thesis, University of Edinburgh, 1992.
28. Sangiorgi, D., The name discipline of uniform receptiveness, *ICALP'97*, LNCS 1256, pp.303–313, 1997.
29. Sekiguchi, T. and Yonezawa, A., A calculus with code mobility, IFIP, pp.21–36, Chapman & Hall, 1997.
30. Sewell, P., Global/Local Subtyping and Capability Inference for a Distributed π-calculus, *ICALP'98*, LNCS 1443, pp.695–706, Springer-Verlag, 1998.
31. Vasconcelos, V. and Honda, K., Principal Typing Scheme for Polyadic π-Calculus. *CONCUR'93*, LNCS 715, pp.524-538, Springer-Verlag, 1993.
32. Vitek, J. and Castagna, G., A Calculus of Secure Mobile Computations, Available at: http://cuiwww.unige.ch/~jvitek, 1999.
33. 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.
34. Yoshida, N., Minimality and Separation Results on Asynchronous Mobile Processes: representability theorems by concurrent combinators. *CONCUR'98*, pp.131–146, LNCS 1466, Springer-Verlag, 1998. Full version as CS Report 05/98, University of Sussex, Available at: http://www.cogs.susx.ac.uk, 1998.
35. The full version of this paper. CS Technical Report 01/99, University of Sussex, Available at: http://www.cogs.susx.ac.uk, 1999.