# A Typed Process Calculus for Fine-Grained Resource Access Control in Distributed Computation

Daisuke Hoshina[1], Eijiro Sumii[2], and Akinori Yonezawa[2]

[1] TOSHIBA Corporation, Japan. hoshina@ivc.toshiba.co.jp
[2] Department of Computer Science, Graduate School of Information Science and Engineering, University of Tokyo, Japan. {sumii,yonezawa}@yl.is.s.u-tokyo.ac.jp

**Abstract.** We propose the $\pi^D$-*calculus*, a process calculus that can flexibly model fine-grained control of resource access in distributed computation, with a type system that statically prevents access violations.

Access control of resources is important in distributed computation, where resources themselves or their contents may be transmitted from one domain to another and thereby vital resources may be exposed to unauthorized processes. In $\pi^D$, a notion of hierarchical *domains* is introduced as an abstraction of protection domains, and considered as the unit of access control. Domains are treated as first-class values and can be created dynamically. In addition, the hierarchal structure of domains can be extended dynamically as well. These features are the source of the expressiveness of $\pi^D$. This paper presents the syntax, the operational semantics, and the type system of $\pi^D$, with examples to demonstrate its expressiveness.

## 1  Introduction

*Background.* Keeping access to resources under control is an issue of central importance in distributed computation: by definition, a distributed system consists of multiple computation domains, such as separate Java virtual machines [8] on different computers, where resources are transmitted from one domain to another; accordingly, a non-trivial amount of effort needs to be spent on ensuring that vital resources are protected from unauthorized access. Although several foundational calculi [3, 7, 10, 22] have been proposed for the purpose of studying distributed computation, few of them have notions of access rights and can guarantee properties about resource access—such as "this high-level integer is read only by high-level processes"—and none of them are flexible enough to allow various policies of access control.

*The $\pi^D$-Calculus.* To address the above issue, we propose the $\pi^D$-*calculus*, a typed process calculus that can flexibly model fine-grained resource access control in distributed computation, where the type system statically prevents access violations. $\pi^D$ has a notion of *domains*, an abstraction of hierarchical protection domains as the unit of access control. For example, consider a situation

where only high-level processes, such as a super user process, can access high-level resources, such as the file /etc/passwd. This situation can be modeled as a system with a communication channel $readPasswd$ of type $\text{chan}\langle high\_level, high\_level \rangle String$. Here, a channel type $\text{chan}\langle i, o \rangle T$ means that channels of this type can be used for *receiving* a value of type $T$ only by processes running in the domain $i$ or a greater domain, and for *sending* a value of type $T$ only by processes running in the domain $o$ or a greater domain. Thus, the system $high\_level[readPasswd?(x).P]$, which represents a process running at the domain $high\_level$ and trying to read a value $x$ from the channel $readPasswd$, is well-typed provided that $P$ is well-typed at $high\_level$, while the system $low\_level[readPasswd?(x).Q]$ is not.

The role of our domains is similar to that of Riely and Hennessy's *security levels* [11]. Unlike their security levels, however, *domains can be created dynamically and are treated as first-class values.* This feature is the source of the expressiveness of $\pi^D$, as demonstrated by the following examples.

*Dynamic Creation of Domains.* First, we give examples that take advantage of the dynamic creation of domains. Consider a situation where there are some CGI programs in a web server. The web server receives a request from a client and executes the corresponding CGI program in the domain *user*, so that the CGI program cannot access vital resources in the web server. This situation can be modeled as follows

$$server[*req_1?(x).\text{spawn}@user.\mathbf{CGI_1} \mid *req_2?(x).\text{spawn}@user.\mathbf{CGI_2} \mid \cdots]$$

where the process $\mathbf{CGI}_i$ represents each CGI program in the web server. The construct $\text{spawn}@user.\mathbf{CGI}_i$ represents the spawning of the process $\mathbf{CGI}_i$ in the domain *user*. After receiving requests, the above process will evolve to a process like

$$server[*req_1?(x).\text{spawn}@user.\mathbf{CGI_1} \mid *req_2?(x).\text{spawn}@user.\mathbf{CGI_2} \mid \cdots] \mid$$
$$user[\mathbf{CGI'_1}] \mid user[\mathbf{CGI'_2}] \mid \cdots$$

where the process $\mathbf{CGI'_i}$ represents an instance of the CGI program $\mathbf{CGI}_i$. Suppose that $\mathbf{CGI'_1}$ has vital resources which *should not* be accessed by other CGI programs (or other instances of the same CGI program). Suppose furthermore that $\mathbf{CGI'_2}$ may be a malicious process trying to interfere with other processes through shared resources in the environment. Since $\mathbf{CGI_2}$ is a process running in the same domain *user* as $\mathbf{CGI_1}$ is running in, it *is* actually possible for $\mathbf{CGI_2}$ to access the vital resources of $\mathbf{CGI_1}$.

A naive solution to this problem would be to prepare another domain $user_2$, in which the web server executes $\mathbf{CGI_2}$. However, doing so means that if every $\mathbf{CGI}_i$ is malicious, we must prepare a distinct domain for each $\mathbf{CGI}_i$ *in advance*. Furthermore, even doing so does not protect different instances of the same CGI program from one another.

These problems can be solved by taking advantage of the *dynamic creation* of domains and rewriting the above process as follows.

$$server[*req_1?(x).(\nu user : \mathtt{dom}\langle server/\bot\rangle)\mathtt{spawn}@user.\mathbf{CGI_1} \;|$$
$$*req_2?(x).(\nu user : \mathtt{dom}\langle server/\bot\rangle)\mathtt{spawn}@user.\mathbf{CGI_2} \;|\; \cdots]$$

The construct $(\nu user : T)$ denotes the dynamic creation of the domain *user* of type $T$. The types of domains have the form $\mathtt{dom}\langle \bar{m}/\bar{n}\rangle$, which determines a hierarchical structure of domains: if a domain $l$ has type $\mathtt{dom}\langle \bar{m}/\bar{n}\rangle$, then $l$ is a child domain of each $m_i$ in $\bar{m}$ and a parent domain of each $n_j$ in $\bar{n}$; that is, $l$ is less than each $m_i$ and greater than each $n_j$. We assume that there exist the greatest domain $\top$ and the least domain $\bot$.

The above solution—that is, creating a fresh domain for each request and executing the requested CGI program in the fresh domain—amounts to *sandboxing* each instance of the CGI program within a separate domain. In practice, however, it is too restrictive to disallow CGI programs to share *any* resources. Let us consider a more flexible policy where there is some public resource—the standard Perl library, for example—which can be accessed by any CGI programs. This policy can be described as follows.

$$(\nu public : \mathtt{dom}\langle server/\bot\rangle)(\nu perl\_library : \mathtt{chan}\langle server, public\rangle T)$$
$$server[perl\_library?(x).P \;|$$
$$*req_1?(x).(\nu user : \mathtt{dom}\langle server/public\rangle)\mathtt{spawn}@user.\mathbf{CGI_1} \;|$$
$$*req_2?(x).(\nu user : \mathtt{dom}\langle server/public\rangle)\mathtt{spawn}@user.\mathbf{CGI_2} \;|\; \cdots]$$

Here, the channel *perl_library* is the resource which can be used by any CGI programs. Note that not only domains but also their hierarchical structure is created dynamically: each instance of *user* is declared to be less than *server* and greater than *public*.

*Domains as First-Class Values.* Now, we give an example where domains are treated as first-class values. Consider a situation where a server receives a request from a client and creates a library which is supposed to be used only by the client. This situation can be modeled as a system **Server** | **Client₁** | **Client₂** | **Client₃** | $\cdots$, where

$$\mathbf{Server} = server[*req?(x).(\nu lib : \mathtt{dom}\langle x, server/\bot\rangle)\mathtt{spawn}@lib.\mathbf{Lib} \;|\; \cdots]$$
$$\mathbf{Client}_i = client_i[req!\langle client_i\rangle \;|\; P_i]$$

Here, the process **Lib** represents the library. The server first receives a domain $client_i$, creates a fresh child domain $lib$, and then runs the library in this domain. Thus, the above system evolves to

**Server** | $client_1[P_1]$ | $lib_1[\mathbf{Lib}]$ | $client_2[P_2]$ | $lib_2[\mathbf{Lib}]$ | $client_3[P_3]$ | $lib_3[\mathbf{Lib}]$ | $\cdots$

where the domain $lib_i$ is less than both *server* and $client_i$. The server does not know the domain $client_i$ of each client in advance. A client, however, informs the server of the client's own domain $client_i$ by sending it to the server through the channel *req*, so that the server can execute the library in the client's child domain $lib_i$.

**Table 1.** Meta-variables for names

$$a - d \in \mathit{Chan} \qquad l - n \in \mathit{Dom} \supset \{\top, \bot\} \qquad u - w \in \mathit{Name} = \mathit{Chan} \cup \mathit{Dom}$$

*Outline of the Paper.* The rest of this paper is organized as follows. Section 2 presents the syntax and the operational semantics of $\pi^D$. Then, Section 3 describes our type system for controlling access to resources in $\pi^D$, and Section 4 proves the type system sound with respect to the operational semantics. Furthermore, Section 5 extends $\pi^D$ with subtyping. Finally, Section 6 discusses related work and Section 7 concludes with future work.

## 2 The Language

This section presents the syntax and the operational semantics of the $\pi^D$-calculus. It is an extension of the polyadic $\pi$-calculus [15] with hierarchical domains, the unit of access control. Although they are similar to security levels of Hennessy and Riely [11], our domains can be created dynamically and passed through communication channels. Every process is located in a domain, which determines which resources the process can access.

Domains are partially ordered. We assume that there exist the greatest domain $\top$ and the least domain $\bot$. Intuitively, processes in a domain which is greater with respect to the partial order can access more resources. We will come back to this point in Section 3.1.

### 2.1 Syntax

We assume that there are two disjoint countably infinite sets *Chan* of *channels* and *Dom* of *domains*. We also assume that the set *Dom* has two special elements $\top$ and $\bot$. Our meta-variable convention for elements of these sets is given in Table 1.

**Types.** We introduce three kinds of types: *channel types*, *domain types* and *dependent pair types*. They are given in Table 2.

**Channel types:** $\mathtt{chan}\langle m, n\rangle T$ is the type of a channel for communicating compound names of type $T$. The pair $\langle m, n \rangle$ means that this channel can be used for input (resp. output) only by processes located in the domain $m$ (resp. $n$) or a greater domain. If $m$ is $\top$, no process can use this channel for input and if $m$ is $\bot$, any process can use this channel for input. The case for output is similar when $n$ is either $\top$ or $\bot$. When a channel $c$ has type $\mathtt{chan}\langle m, n\rangle T$, we say that $c$ has the input level $m$ and the output level $n$.

**Domain types:** If a domain $l$ has type $\mathtt{dom}\langle \bar{m}/\bar{n}\rangle$, then $l$ is a child domain of each $m$ in $\bar{m}$ and a parent domain of each $n$ in $\bar{n}$. That is, $l$ is less than any of $\bar{m}$ and greater than any of $\bar{n}$.

**Dependent pair types:** If a name $v$ has type $S$ and another name $v'$ has type $\{v/u\}T$, then the pair $(v, v')$ has a dependent pair type $\Sigma u : S.T$. Note that if $u$ does not appear free in $T$, this dependent pair type can actually be considered as an ordinary pair type $S \times T$.

**Processes.** The syntax of *threads* and *systems* is given in Table 3. Both threads and systems are called *processes*. Their intuitive meanings are as follows.

– An *output* $c!\langle V \rangle$ sends the compound name $V$ on the channel $c$. An *input* $c?(U : T).P$ receives a compound name through the channel $c$, binds the compound name to $U$, and then executes $P$.
– A *name creation* $(\nu v : T)P$ or $(\nu v : T)M$ creates a fresh name, binds it to $v$, and then executes $P$ or $M$. Note that not only channels but also domains can be created dynamically.
– A *spawning* $\texttt{spawn@}m.P$ spawns the process $P$ in the domain $m$.
– A *located thread* $m[P]$ denotes the thread $P$ running in the domain $m$.

We write $\textit{fn}(T)$, $\textit{fn}(U)$, $\textit{fn}(P)$ and $\textit{fn}(M)$ for the set of free names appearing in $T$, $U$, $P$ and $M$, respectively. Their formal definition is omitted in this paper.

**Table 4.** Structural preorder

$$
\begin{array}{lll}
m[P \mid Q] \preceq m[P] \mid m[Q] & & \text{(SP-SPLIT)} \\
m[(\nu v : T)P] \preceq (\nu v : T)m[P] & \text{if } v \neq m & \text{(SP-NAME)} \\
m[0] \preceq 0 & & \text{(SP-ZERO)} \\
(\nu v : T)M \mid N \preceq (\nu v : T)(M \mid N) & \text{if } v \notin fn(N) & \text{(SP-EXTR)} \\
m[*P] \preceq m[P] \mid m[*P] & & \text{(SP-REPL)} \\
(\nu v : T)(\nu w : S)M \preceq (\nu w : S)(\nu v : T)M & \text{if } v \notin fn(S) \wedge w \notin fn(T) & \text{(SP-EX)}
\end{array}
$$

## 2.2 Operational Semantics

We define the operational semantics of $\pi^D$ by using two binary relations: the *structural relation* $\preceq$ and the *reduction relation* $\longrightarrow$.

**Definition 1 (Structural Relation).** *The structural relation $\preceq$ is the least reflexive and transitive relation over systems satisfying the rules in Table 4 and the monoid laws: $M \mid 0 \equiv M, M \mid N \equiv N \mid M$ and $(L \mid M) \mid N \equiv L \mid (M \mid N)$. Here, $P \equiv Q$ is defined as $P \preceq Q \wedge Q \preceq P$.*

The rule (SP-SPLIT) allows a thread $P \mid Q$ to split into two independent threads $P$ and $Q$. (SP-EX) says that adjacent name bindings can be swapped. Note that the side-condition is necessary since we have dependent pair types [22].

**Definition 2 (Reduction Relation).** *The reduction relation $\longrightarrow$ over systems is the least relation satisfying the rules in Table 5. $M \longrightarrow M'$ means that $M$ can evolve to $M'$ by one-step communication or spawning. The relation $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.*

The rule (R-COMM) allows two processes running in parallel to exchange a compound name through a channel. Be aware that the communication can take place across domains. Thus, unlike in $D\pi$ [10], channel names are global in $\pi^D$. The rule (R-SPAWN) allows a process to spawn a thread in a domain. Obviously, it is dangerous if any process can spawn a new process in any domain by using the spawn statement. Our type system will guarantee that processes in a domain $m$ can spawn new processes in another domain $n$ only if $m$ is greater than $n$. The other rules are standard in the $\pi$-calculus.

## 2.3 Example

*Computation Server.* We give an example to show how we protect channels from unauthorized access by assigning them suitable types. Consider a situation where a computation server receives a request from a client and creates a library which is supposed to be used only by the client, like the last example in Section 1. The library here is an integer successor function: it receives an integer $i$ with a

**Table 5.** Reduction relation

$$m[c!\langle V \rangle] \mid m'[c?(U:T).P] \longrightarrow m'[\{V/U\}P] \quad \text{(R-COMM)}$$

$$m[\mathbf{spawn}@n.P] \longrightarrow n[P] \quad \text{(R-SPAWN)}$$

$$\frac{M \longrightarrow M'}{M \mid N \longrightarrow M' \mid N} \ \text{(R-PAR)} \qquad \frac{M \longrightarrow N}{(\nu v:T)M \longrightarrow (\nu v:T)N} \ \text{(R-NEW)}$$

$$\frac{M \preceq M' \quad M' \longrightarrow N' \quad N' \preceq N}{M \longrightarrow N} \ \text{(R-CONG)}$$

where the substitution $\{V/U\}$ of compound names is defined as follows:

$$\{V/U\} \overset{def}{=} \begin{array}{ll} \{v/u\} & \text{if } V = v \text{ and } U = u \\ (\{v/u\} \bullet \{V'/U'\}) & \text{if } V = (v,V') \text{ and } U = (u,U') \end{array}$$

$$\{v_1/u_1, \cdots, v_i/u_i\} \bullet \{v'_1/u'_1, \cdots, v'_j/u'_j\} \overset{def}{=} \{v_1/u_1, \cdots, v_i/u_i, v'_1/u'_1, \cdots, v'_j/u'_j\}$$

channel $r$, and sends $i+1$ back through $r$. Assuming arithmetic primitives for integer operation, this situation can be modeled as

$$(\nu Serv : \mathbf{dom}\langle \top/\bot \rangle)(\nu Client_1 : \mathbf{dom}\langle \top/\bot \rangle)(\nu Client_2 : \mathbf{dom}\langle \top/\bot \rangle) \cdots$$
$$(\nu serv : \mathbf{TServ})(\mathbf{Serv} \mid \mathbf{Client}_1 \mid \mathbf{Client}_2 \mid \cdots)$$

where

$$\begin{aligned}
\mathbf{Serv} = {}& Serv[*serv?(c,r : \mathbf{TServReq}). \\
& \quad (\nu Succ : \mathbf{dom}\langle Serv/\bot \rangle)(\nu succ : \mathbf{TSucc}_{c,Succ}) \\
& \quad (\mathbf{spawn}@Succ.\mathbf{Succ} \mid r!\langle Succ, succ \rangle)] \\
\mathbf{Succ} = {}& *succ?(x,y : \mathbf{TSuccReq}_{c,Succ}).y!\langle x+1 \rangle \\
\mathbf{Client}_i = {}& Client_i[(\nu reply : \mathbf{chan}\langle Client_i, Serv \rangle \mathbf{TServAns}_{Client_i}) \\
& \quad serv!\langle Client_i, reply \rangle \mid \\
& \quad reply?(MySucc, mysucc : \mathbf{TServAns}_{Client_i}). \\
& \quad \ldots \text{use } mysucc \text{ as a library} \ldots)]
\end{aligned}$$

$$\begin{aligned}
\mathbf{TServ} &= \mathbf{chan}\langle Serv, \bot \rangle \mathbf{TServReq} \\
\mathbf{TServReq} &= \Sigma x : \mathbf{dom}\langle \top/\bot \rangle.\mathbf{chan}\langle x, Serv \rangle \mathbf{TServAns}_x \\
\mathbf{TServAns}_m &= \Sigma y : \mathbf{dom}\langle Serv/\bot \rangle.\mathbf{TSucc}_{m,y} \\
\mathbf{TSucc}_{m,n} &= \mathbf{chan}\langle n, m \rangle \mathbf{TSuccReq}_{m,n} \\
\mathbf{TSuccReq}_{m,n} &= \mathbf{int} \times \mathbf{TSuccAns}_{m,n} \\
\mathbf{TSuccAns}_{m,n} &= \mathbf{chan}\langle m, n \rangle \mathbf{int}
\end{aligned}$$

Note that the computation server does not know in advance how many clients exist, and thus every client must send its own domain to the computation server.

**Table 6.** Type environments

$$\Gamma, \Delta ::= \bullet \mid \Gamma, u : T$$

This process evolves to:

$(\nu\,Serv : \mathbf{dom}\langle\top/\bot\rangle)(\nu\,Client_1 : \mathbf{dom}\langle\top/\bot\rangle)(\nu\,Client_2 : \mathbf{dom}\langle\top/\bot\rangle)\cdots$
$(\nu\,serv : \mathbf{TServ})(\mathbf{Serv} \mid$

$\qquad\qquad (\nu\,Succ_1 : \mathbf{dom}\langle Serv/\bot\rangle)(\nu\,succ_1 : \mathbf{TSucc}_{Client_1,Succ_1})$
$\qquad\qquad\quad (\,Client_1[\{Succ_1/MySucc, succ_1/mysucc\}\cdots] \mid Succ_1[\mathbf{Succ}]) \mid$
$\qquad\qquad (\nu\,Succ_2 : \mathbf{dom}\langle Serv/\bot\rangle)(\nu\,succ_2 : \mathbf{TSucc}_{Client_2,Succ_2})$
$\qquad\qquad\quad (\,Client_2[\{Succ_2/MySucc, succ_2/mysucc\}\cdots] \mid Succ_2[\mathbf{Succ}]) \mid$
$\qquad\qquad \cdots)$

The type of the channel *reply* says that, for each $i$, the channel can be used only in $Client_i$ for input and only in $Serv$ for output, so that the communication between $Client_i$ and $Serv$ is never intercepted by other processes. The type of the channel *succ* says that, for each $i$, the channel can be used only in $Serv$ for input and only in $Client_i$ for output, so that only processes in the domain $Client_i$ can invoke the successor function and only processes in the domain $Succ$ can provide the successor function.

## 3 The Type System

The primary judgments of the type system are of the form $\Gamma \vdash P : Th[m]$ for threads and of the form $\Gamma \vdash M : Sys$ for systems, where $\Gamma$ is a *type environment* defined as in Table 6. We write $\Gamma(u) = T$ to mean that $\Gamma$ maps $u$ to $T$, that is, there exist some $\Gamma_1$ and $\Gamma_2$ such that $\Gamma = \Gamma_1, u : T, \Gamma_2$. The domain of $\Gamma$, written $dom(\Gamma)$, is the set of names which $\Gamma$ maps. We write $Dom(\Gamma)$ for the set of names having a domain type in $\Gamma$.

### 3.1 Environments and Partially Ordered Sets of Domains

Type environments determine a partial order among domains. We write $\leq^\Gamma$ for this partial order, whose formal definition is as follows.[1]

**Definition 3 (Partial Order under $\Gamma$).**

$$m \leq^\Gamma n \quad\overset{def}{=}\quad (m = \bot \ \vee\ n = \top \ \vee\ m\,\dot{\leq}^\Gamma n) \ \wedge\ \{m,n\} \subseteq Dom(\Gamma)$$

*where*

$$\dot{\leq}^\Gamma \overset{def}{=} \begin{cases} \emptyset & \text{if } \Gamma = \bullet \\ \dot{\leq}^{\Gamma'} & \text{if } \Gamma = \Gamma', c : \mathbf{chan}\langle m_1, m_2\rangle T \\ \left(\dot{\leq}^{\Gamma'} \cup \{(m,n_i), (n'_j,m) \mid n_i \in \{\bar{n}\}, n'_j \in \{\tilde{n'}\}\}\right)^* \\ & \text{if } \Gamma = \Gamma', m : \mathbf{dom}\langle\bar{n}/\tilde{n'}\rangle \end{cases}$$

---

[1] $\Re^*$ denotes the reflexive and transitive closure of the relation $\Re$.

**Table 7.** Good environments

$$\frac{}{\vdash \bullet \; ok} \; \text{(E-EMPTY)}$$

$$\begin{array}{c} u \notin dom(\Gamma) \cup \{\top, \bot\} \quad \vdash \Gamma \; ok \quad \Gamma \models T \\ \underline{T \text{ is a channel type or a domain type}} \\ \vdash \Gamma, u : T \; ok \end{array} \; \text{(E-TYPE)}$$

$$\begin{array}{c} \forall m_i' \in \{\tilde{m'}\}. \; \forall m_i \in \{\tilde{m}\}. \; m_i' \leq^\Gamma m_i \; \wedge \; m_i' \neq m_i \\ \underline{\{\tilde{m}\} \subseteq Dom(\Gamma) \cup \{\top\} \quad \{\tilde{m'}\} \subseteq Dom(\Gamma) \cup \{\bot\}} \\ \Gamma \models \mathbf{dom}\langle \tilde{m}/\tilde{m'} \rangle \end{array} \; \text{(T-DOM)}$$

$$\frac{m_1, m_2 \in Dom(\Gamma) \cup \{\top, \bot\} \quad \Gamma \models T}{\Gamma \models \mathbf{chan}\langle m_1, m_2 \rangle T} \; \text{(T-CHAN)}$$

$$\frac{\Gamma \models S \quad \Gamma, u : S \models T \quad u \notin dom(\Gamma) \cup \{\top, \bot\}}{\Gamma \models \Sigma u : S.T} \; \text{(T-DEP)}$$

For example, the type environment $m_1{:}\mathbf{dom}\langle\top/\bot\rangle, m_2{:}\mathbf{dom}\langle m_1/\bot\rangle, m_3{:}\mathbf{dom}\langle m_1/m_2\rangle,$ $m_4 : \mathbf{dom}\langle\top/m_3\rangle$ defines the partially ordered set shown in Fig. 1. Note that $m_2 \leq^\Gamma m_4$ also holds by transitivity.
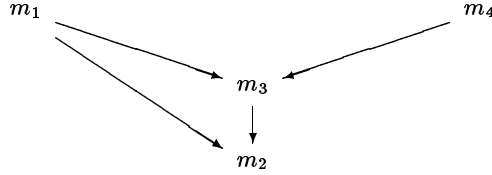


**Fig. 1.** Example of domain hierarchy. The domains $\top$ and $\bot$ are omitted.

The formation rules for environments, which are of the form $\vdash \Gamma \; ok$, are given in Table 7. They are defined by using the formation rules for channel types and domain types, which are of the form $\Gamma \models T$. The rule (T-DOM) deserves mention. The premise in the first line ensures that parents of a domain must always be greater than children of the domain. This condition is important for keeping $\leq^\Gamma$ a partial order. The premises in the second line ensure that $\top$ is always the greatest element and $\bot$ is always the least element. The rule (T-DEP) says that we must check $T$ under $\Gamma$ extended with $u : S$, since $u$ may appear free in $T$.

The following proposition assures that if $\Gamma$ is a good environment, $\leq^\Gamma$ is indeed a partial order.

**Proposition 1.** *If $\vdash \Gamma$ ok, then $\leq^\Gamma$ is a partial order over $Dom(\Gamma) \cup \{\top, \bot\}$.*

### 3.2 Typing Rules

**Typing Rules for Threads.** The typing rules for threads in $\pi^D$ are given in Table 8. The judgment $\Gamma \vdash P : Th[l]$ indicates that the thread $P$ is well-typed under the type environment $\Gamma$ and can be located in the domain $l$. Intuitive explanation of key rules is as follows.

**G-DEP** This rule follows from the meaning of compound names of a dependent pair type.

**TH-ZERO** Since a null thread does nothing, it can be located in any domain which is present in the environment.

**TH-OUT** The first and third premises say that $c$ is a channel with the output level $m_2$, and that the compound name $V$ must have the type $T$ of names communicated through the channel. The second premise ensures that $c$ is used for output only by processes in the domain $m_2$ or a greater domain. The forth premise ensures that there is no process located in the domain $\top$.

**TH-IN** This rule is similar to (TH-OUT) except that we must check that the continuation process $P$ is well-typed under $\Gamma$ extended with $U : T$.

**TH-SPAWN** The second premise of this rule ensures that only a thread in a domain $n$ greater than $m$ can spawn a new thread in $m$.

**Typing Rules for Systems.** The judgment $\Gamma \vdash M : Sys$ indicates that the system $M$ is well-typed under the type environment $\Gamma$. The typing rules for systems are also given in Table 8. The rules (SYS-ZERO), (SYS-NEW) and (SYS-PAR) are similar to those for threads. The rule (SYS-EX) allows adjacent elements of a type environment to be switched. This rule is necessary only for a technical reason, that is, for proving the case of (SP-EX) in Lemma 2 in Section 4. The rule (SYS-LOCATED) says that only threads of type $Th[m]$ can be located in the domain $m$.

Well-typedness of processes is obviously decidable since the typing rules are purely syntax-directed and the partial order among domains is finite.

## 4 Type Soundness

### 4.1 Subject Reduction

Our first main result is the subject reduction theorem. In order to prove that, we need the following lemmas:

**Table 8.** Typing rules for threads and systems

---

$$\Gamma + U : T \stackrel{def}{=} \Gamma + u : T \qquad\qquad \text{if } U : T = u : T$$
$$(\Gamma, u : S) + U' : \{u/u'\}T' \text{ if } X : T = (u, U') : (\Sigma u' : S.T')$$

$$\frac{\vdash \Gamma \; ok \quad \Gamma(u) = T}{\Gamma \vdash u : T}\text{(G-NAME)} \qquad \frac{\Gamma \vdash v : S \quad \Gamma \vdash V : \{v/u\}T}{\Gamma \vdash (v, V) : (\Sigma u : S.T)}\text{(G-DEP)}$$

$$\frac{\Gamma \vdash c : \mathbf{chan}\langle m_1, m_2 \rangle T \quad m_2 \leq^\Gamma l \quad \Gamma \vdash V : T \quad l \neq \top}{\Gamma \vdash c!\langle V \rangle : Th[l]}\text{(TH-OUT)}$$

$$\frac{\begin{array}{c}\Gamma \vdash c : \mathbf{chan}\langle m_1, m_2 \rangle T \quad m_1 \leq^\Gamma l \quad l \notin fn(U) \\ \Gamma + U : T \vdash P : Th[l] \quad l \neq \top\end{array}}{\Gamma \vdash c?(U : T).P : Th[l]}\text{(TH-IN)}$$

$$\frac{\Gamma \vdash P : Th[m] \quad m \leq^\Gamma n}{\Gamma \vdash \mathbf{spawn}@m.P : Th[n]}\text{(TH-SPAWN)} \qquad \frac{\Gamma \vdash P : Th[l]}{\Gamma \vdash *P : Th[l]}\text{(TH-REP)}$$

$$\frac{\vdash \Gamma \; ok \quad l \in Dom(\Gamma)}{\Gamma \vdash 0 : Th[l]}\text{(TH-ZERO)} \qquad \frac{\Gamma, v : T \vdash P : Th[l] \quad v \neq l}{\Gamma \vdash (\nu v : T)P : Th[l]}\text{(TH-NEW)}$$

$$\frac{\Gamma \vdash P : Th[l] \quad \Gamma \vdash Q : Th[l]}{\Gamma \vdash P \mid Q : Th[l]}\text{(TH-PAR)}$$

$$\frac{\vdash \Gamma \; ok}{\Gamma \vdash 0 : Sys}\text{(SYS-ZERO)} \qquad \frac{\Gamma, v : T \vdash M : Sys}{\Gamma \vdash (\nu v : T)M : Sys}\text{(SYS-NEW)}$$

$$\frac{\Gamma \vdash P : Th[m]}{\Gamma \vdash m[P] : Sys}\text{(SYS-LOCATED)} \qquad \frac{\Gamma \vdash M : Sys \quad \Gamma \vdash N : Sys}{\Gamma \vdash M \mid N : Sys}\text{(SYS-PAR)}$$

$$\frac{\Gamma_1, u_1 : S, u_2 : T, \Gamma_2 \vdash M : Sys \quad u_1 \notin fn(T) \quad u_2 \notin fn(S)}{\Gamma_1, u_2 : T, u_1 : S, \Gamma_2 \vdash M : Sys}\text{(SYS-EX)}$$

---

**Lemma 1 (Substitution).**

- If $\Gamma, u : T, \Gamma' \vdash P : Th[n]$ and $\Gamma, \{v/u\}\Gamma' \vdash v : T$, then $\Gamma, \{v/u\}\Gamma' \vdash \{v/u\}P : \{v/u\}Th[n]$.
- If $\Gamma + X : T \vdash P : Th[n]$ and $\Gamma \vdash V : T$, then $\Gamma \vdash \{V/X\}P : \{V/X\}Th[n]$.

**Lemma 2 (Subject Preordering).** If $\Gamma \vdash M : Sys$ and $M \preceq N$, then $\Gamma \vdash N : Sys$.

In $\pi^D$, domains appear free in types, so the substitution lemmas above are a little different from usual ones: the subjects of substitution are not only free names of $P$ but also those of $\Gamma'$ and $Th[n]$.

**Table 9.** Tagged systems

---

(**Tagged Systems**)
$E, F, G \qquad \qquad ::= \ldots$
$\qquad \qquad \qquad \qquad m[P]^{\tilde{n}}$ tagged located thread

---

**Theorem 1 (Subject Reduction).** *If $\Gamma \vdash M : Sys$ and $M \longrightarrow^* N$, then $\Gamma \vdash N : Sys$.*

*Proof.* Follows from Lemma 1 and Lemma 2. See the full version [12] for details.
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

This theorem says that well-typedness of processes is preserved throughout computation.

### 4.2 The Tagged Language and Type Safety

Roughly, our type safety theorem says that:

> A channel $c$ of type $\text{chan}\langle m_1, m_2 \rangle T$ can be used for output only by processes which were located in $m_2$ or a greater domain *at the beginning of the computation*. (The case for input is similar and omitted.)

In order to state and prove this theorem, we introduce a tagged language. Each process of the tagged language carries its own tag, which records the history of domains it has moved across. Note that tags are introduced only for stating properties of processes and do not affect their execution.

**The Tagged Language.**

*Tagged Systems.* The formal definition of tagged systems is given in Table 9. Systems in the tagged language are the same as in the original language except that located threads are tagged with a sequence of domains. The tag $\tilde{n}$ of a located thread $P_1$ represents the domain in which $P_1$ was spawned, the domain in which the parent thread $P_2$ which spawned $P_1$ was spawned, the domain in which the grand parent thread $P_3$ which spawned $P_2$ was spawned, and so forth. For example, a thread which is located in $m_1$ now, whose parent thread was located in $m_2$, and whose grand parent thread was located in $m_3$, is represented as $m_1[P]^{m_1, m_2, m_3}$.

*Operational Semantics.* The operational semantics of the tagged language is given in Table 10. The definition of the structural preorder is a straightforward extension of the original. We show only (TS-SPLIT) and (TS-NAME). As for the reduction relation, only (R-SPAWN) is changed so that when a new thread is spawned in a domain, it is recorded in the thread's tag.

**Table 10.** Structural preorder and reduction relation

$$m[P \mid Q]^{\tilde{n}} \preceq m[P]^{\tilde{n}} \mid m[Q]^{\tilde{n}} \qquad\qquad \text{(TS-SPLIT)}$$
$$m[(\nu v : T)P]^{\tilde{n}} \preceq (\nu v : T)m[P]^{\tilde{n}} \text{ if } v \notin \{m, \tilde{n}\} \quad \text{(TS-NAME)}$$

$$m[\mathbf{spawn}@n.P]^{\tilde{n}'} \longrightarrow n[P]^{n, \tilde{n}'} \qquad\qquad\qquad \text{(TR-SPAWN)}$$

*Technical Properties of Tagged Language.* We present two lemmas stating that the semantics of a program does not depend on whether it is written in the original language or in the tagged language.

For convenience, we define a function *Tag*, which maps terms of the original language to terms of the tagged language, and *Erase*, which is the inverse of *Tag*.

**Definition 4 (Tagging).**

$$\begin{aligned}
Tag(M' \mid N') &= Tag(M') \mid Tag(N') \\
Tag((\nu v : T)M') &= (\nu v : T)\,Tag(M') \\
Tag(0) &= 0 \\
Tag(n[P]) &= n[P]^n
\end{aligned}$$

**Definition 5 (Erase).** *Erase(_) is the function mapping tagged system expressions to ordinary system expressions by erasing tags.*

**Lemma 3 (Correspondence w.r.t $\preceq$).**

1. *If $E \preceq F$, then $Erase(E) \preceq Erase(F)$.*
2. *If $Erase(E) \preceq M$, then there is some $F$ such that $E \preceq F$ and $M = Erase(F)$.*

**Lemma 4 (Correspondence w.r.t $\longrightarrow$).**

1. *If $E \longrightarrow F$, then $Erase(E) \longrightarrow Erase(F)$.*
2. *If $Erase(E) \longrightarrow M$, then there is some some $E'$ such that $E \longrightarrow E'$ and $M = Erase(E')$.*

**Type Safety.** Before describing the formal definition of our type safety theorem, we make some preparations for proving it. First, we define the notion of *guarded* threads and *guarded tagged systems*, used in the lemma below.

**Definition 6 (Guarded Threads and Guarded Tagged Systems).** *A thread is guarded if it is an input, output, replication, spawning or null. A tagged system is guarded if it is a located thread or null.*

The following lemma says that each domain in the history of a well-typed system is less than the previous domain.

**Lemma 5.** *Suppose $\Gamma \vdash M : Sys$ and $E = Tag(M)$. If there is some $F$ such that $E \longrightarrow^* F \preceq (\nu\Gamma')(F_1 \mid \cdots \mid F_q)$ where $F_1 \cdots F_q$ are guarded, then $m^i = m_1^i \leq^{\Gamma,\Gamma'} \ldots \leq^{\Gamma,\Gamma'} m_{p^i}^i$ holds for every $F_i = m^i[P_i]^{m_1^i,\ldots,m_{p^i}^i}$.*

From the above lemma and the type system, the following lemma can be proved. This lemma guarantees that, for example, a channel having the output level $n_2$ is used for output only by processes whose history contains only domains greater than $n_2$.

**Lemma 6.** *Suppose $\Gamma \vdash M : Sys$ and $E = Tag(M)$. If there is some $F$ such that $E \longrightarrow^* F \preceq (\nu\Gamma')(m'[c!\langle V\rangle]^{m_1',\ldots,m_i'} \mid F')$ and $(\Gamma,\Gamma')(c) = \mathtt{chan}\langle n_1, n_2\rangle T$, then $n_2 \leq^{\Gamma,\Gamma'} m' = m_1' \leq^{\Gamma,\Gamma'} \ldots \leq^{\Gamma,\Gamma'} m_i'$.[2]*

Furthermore, in order to prove the type safety theorem, we need the following lemma. This lemma says that, for example, if $m \leq^{\Gamma, l:\mathtt{dom}\langle n/m\rangle} n$, then $m \leq^{\Gamma} n$ even though $l : \mathtt{dom}\langle n/m\rangle$ is removed from $\Gamma, l : \mathtt{dom}\langle n/m\rangle$.

**Lemma 7.** *Suppose $\vdash \Gamma, \Gamma'$ ok and $m \leq^{\Gamma,\Gamma'} n$. If $m, n \in dom(\Gamma)$, then $m \leq^{\Gamma} n$.*

Finally, we prove the type safety theorem. Its formal statement is as follows.

**Theorem 2 (Type Safety).** *Suppose $\Gamma \vdash M : Sys$, $E = Tag(M)$ and $\Gamma, \Gamma' = \Delta, c : \mathtt{chan}\langle n_1, n_2\rangle T, \Delta'$. Suppose also $M \longrightarrow^* N \preceq (\nu\Gamma')(m'[c!\langle V\rangle] \mid N')$. Then, there exist some $F$, $F'$, and $\tilde{m}'$ such that $E \longrightarrow^* F \preceq (\nu\Gamma')(m'[c!\langle V\rangle]^{m_1',\ldots,m_k'} \mid F')$ and $N = Erase(F)$. Furthermore, $n_2 \leq^{\Gamma,\Gamma'} m' = m_1' \leq^{\Gamma,\Gamma'} \ldots \leq^{\Gamma,\Gamma'} m_k'$ where $n_2 \leq^{\Delta} m_j'$ for every $m_j' \in dom(\Delta)$.*

*Proof.* Immediately follows from Lemma 4, Lemma 6 and Lemma 7. □

Intuitively, this statement means that if a process uses the channel $c$ for output, then:

1. each domain in the history of the process (i.e., $m_1', \cdots, m_k'$) must be greater than $n_2$, and
2. for each $j$, if $m_j'$ was created before $c$ (i.e., $m_j' \in dom(\Delta)$), then $m_j'$ was *originally* greater than $n_2$.

In other words, processes which were not located in $n_2$ or a greater domain at the beginning of the execution of $M$ can never use $c$ for output throughout computation.

We give two examples to demonstrate how the typing rules in Section 3 prevents failure of type safety. Suppose $\Gamma = m : \mathtt{dom}\langle\top/\bot\rangle, n : \mathtt{dom}\langle m/\bot\rangle, c : \mathtt{chan}\langle\top, m\rangle\mathtt{int}$.

- First, consider a process $P \stackrel{def}{=} n[c!\langle 1\rangle]$ trying to use $c$, a channel with an output level $m$, for output within $n$. Since the tagged version of $P$ is $n[c!\langle 1\rangle]^n$ and $m \not\leq^{\Gamma} n$, the first claim above of the theorem does not hold for $P$. By (TH-OUT), however, $P$ cannot be well-typed, since $m$ is not greater than or equal to $n$.

---
[2] $(\nu\Gamma)$ is an abbreviation of $(\nu u_1 : T_1)\cdots(\nu u_n : T_n)$ for $\Gamma = u_1 : T_1, \cdots, u_n : T_n$.

**Table 11.** Subtyping relation

$$\frac{\begin{array}{c} m \leq^\Gamma m' \quad m' \neq \top \Rightarrow \Gamma \vdash S \sqsubseteq T \\ n \leq n' \quad n' \neq \top \Rightarrow \Gamma \vdash T \sqsubseteq S \end{array}}{\Gamma \vdash \mathtt{chan}\langle m, n \rangle S \sqsubseteq \mathtt{chan}\langle m', n' \rangle T} \text{(SUB-CHAN)}$$

$$\frac{\{\tilde{m}'\} \subseteq \{\tilde{m}\} \subseteq Dom(\Gamma) \quad \{\tilde{n}'\} \subseteq \{\tilde{n}\} \subseteq Dom(\Gamma)}{\Gamma \vdash \mathtt{dom}\langle \tilde{m}/\tilde{n} \rangle \sqsubseteq \mathtt{dom}\langle \tilde{m}'/\tilde{n}' \rangle} \text{(SUB-DOM)}$$

$$\frac{\Gamma, u : S \vdash T \sqsubseteq T' \quad \Gamma \vdash S \sqsubseteq S' \quad u \text{ is fresh}}{\Gamma \vdash \Sigma u : S.T \sqsubseteq \Sigma u : S'.T'} \text{(SUB-DEP)}$$

- Second, consider a more cunning process $Q \stackrel{def}{=} n[(\nu l{:}\mathtt{dom}\langle n/m \rangle)\mathtt{spawn}@l.c!\langle 1 \rangle]$, which tries to create a new domain $l$ as a *parent* of the domain $m$, and then spawn a process trying to use the channel $c$ for output. Although the first claim of the theorem *does* hold for $Q$, the second does not for the following reason: the tagged version of $P$ is $n[(\nu l : \mathtt{dom}\langle n/m \rangle)\mathtt{spawn}@l.c!\langle 1 \rangle]^n$ which evolves to $(\nu l : \mathtt{dom}\langle n/m \rangle)l[c!\langle 1 \rangle]^{l,n}$; indeed $m \leq^{\Gamma, l:\mathtt{dom}\langle n/m \rangle} l \leq^{\Gamma, l:\mathtt{dom}\langle n/m \rangle} n$ holds, but $m \leq^\Gamma n$ does not. By (T-DOM), however, $Q$ cannot be well-typed either, since $n$ is not greater than or equal to $m$.

## 5 Subtyping

The partial order among domains induces a subtyping relation on types. Intuitively, the subtyping relation $\Gamma \vdash S \sqsubseteq T$ says that a name of type $S$ may be used as a name of type $T$. A possible definition of the subtyping relation is given in Table 11.

**SUB-CHAN** The first premise follows from the intuition that a channel with input level $m$ can be used by threads in a domain $m'$ greater than or equal to $m$. The third premise is similar. The second and forth premises say that an input capability implies covariance of the channel type, whereas an output capability implies contravariance [16].

**SUB-DOM** This rule follows from the intuition that a child (resp. parent) domain of both $m$ and $n$ can be used as a child (resp. parent) domain of only $m$.

**SUB-DEP** This rule can be considered as an extension of the usual definition of subtyping relation for pair types. The difference is that we must check $T \sqsubseteq T'$ under $\Gamma$ extended with $u : S$.

Furthermore, the typing rules for threads can be refined as in Table 12.

**TH-OUT** The first, third, and fourth premises say that $V$ must be of less restrictive type than the type $T$ of names communicated through the channel.

**Table 12.** Typing rules for threads with subtyping

$$\frac{\begin{array}{c} \Gamma \vdash c : \mathtt{chan}\langle m_1, m_2 \rangle T \quad m_2 \leq^{\Gamma} l \\ \Gamma \vdash T' \sqsubseteq T \quad \Gamma \vdash V : T' \quad l \neq \top \end{array}}{\Gamma \vdash c!\langle V \rangle : Th[l]} \text{(TH-OUT)}$$

$$\frac{\begin{array}{c} \Gamma \vdash c : \mathtt{chan}\langle m_1, m_2 \rangle T \quad m_1 \leq^{\Gamma} l \quad l \notin \mathit{fn}(U) \\ \Gamma \vdash T \sqsubseteq T' \quad \Gamma + U : T' \vdash P : Th[l] \quad l \neq \top \end{array}}{\Gamma \vdash c?(U : T').P : Th[l]} \text{(TH-IN)}$$

**TH-IN** The first, forth, and fifth premises say that compound names received through $c$ can be regarded as names of more restrictive type than $T$.

Although the definitions in Table 11 and Table 12 seem reasonable, we have not yet proved soundness of the type system with this subtyping relation.

# 6 Related Work

We have already made reference to Riely and Hennessy's work [10, 11]. Besides it, several other foundational calculi have been proposed for access control in distributed computation.

- Yoshida and Hennessy [22] proposed a type system for access control in the higher-order $\pi$-calculus. Their type system can be used for controlling access to resources in higher-order code transmitted from one domain to another. Introducing our notion of domains into their calculus might enable finer-grained access control in the higher-order $\pi$-calculus.
- De Nicola, Ferrari and Pugliese [5] also studied access control in distributed computation. They dealt with a variant of Linda with multiple tuple spaces as a target language. Tuple spaces correspond to domains in $\pi^D$, and tuples (named data) correspond to resources. Since the type system in [5] controls access to tuple spaces rather than to specific tuples, it provides coarser-grained access control than [10] and $\pi^D$.
- Dezani-Ciancaglini and Salvo [6] introduced *security levels* [11] into the ambient calculus [3]. Their security levels are associated with ambients: it is guaranteed that an ambient at security level $s$ can be traversed or opened only by ambients at security level $s$ or greater. The role of security levels are similar to that of our domains. They, however, are not first-class values and cannot be created dynamically. Again, adapting our notion of domains for the ambient calculus might enable more flexible access control.
- Cardelli and Gordon [4] introduced *groups* into the $\pi$-calculus. The role of groups is similar to that of domains. Groups can be created dynamically, but are not first-class values.

Foundational calculi for studying various security issues have also been proposed by several authors [1, 2, 9, 21]. Abadi and Gordon [1] proposed the spi calculus, an extension of the $\pi$-calculus with cryptographic primitives. It allows programmers to describe and analyze security protocols. Bugliesi and Castagna [2] proposed a typed variant of safe ambient [14]. Their type system can capture not only explicit but also implicit behavior of processes and thereby detects security attacks such as Trojan horses. Heintze and Riecke [9] proposed an extension of the $\lambda$-calculus with a type system for preserving secrecy and integrity of resources. Vitek and Castagna [21] proposed a low level language, called the seal calculus, for writing secure distributed applications over a large-scale open network.

Our notion of domains is orthogonal to the notion of locations [7, 10, 20]: locations are targets of code movement, while domains are the unit of access control; it is possible to have different domains within a single location, or to have a single domain across multiple locations.

## 7 Future Work

Our type safety theorem holds if and only if all processes are well-typed. This assumption is quite restrictive since it is not realistic to verify open systems, such as the Internet, as a whole [19]. This limitation would be overcome by adapting type systems where a certain type safety theorem holds even if some of the processes are not well-typed [4, 19].

Types constrain the behavior of processes and their environments, and thereby allow a coarser notion of behavioral equivalence. Typed equivalence has already been investigated in various process calculi [13, 16–18]. It may be possible to develop a similar theory in $\pi^D$.

## Acknowledgment

## References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997.

[2] M. Bugliesi and G. Castagna. Secure safe ambients. In *POPL'01*. ACM Press, 2001.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In *FoSSaCS'98*. Springer-Verlag, 1998.

[4] L. Cardelli and A. D. Gordon. Secrecy and group creation. In *CONCUR'2000*. Springer-Verlag, 2000.

[5] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim : a kernel language for agents interaction and mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.

[6] M. Dezani-Ciancaglini and I. Salvo. Security types for mobile safe ambients. In *ASIAN'00*, 2000.

[7] C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *CONCUR'96*. Springer-Verlag, 1996.

[8] J. Gosling, B. Joy, and G. Steele. *The Java language specification*. Addison-Wesley, 1996.

[9] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *POPL'98*. ACM Press, 1998.

[10] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *HLCL'98*. Elsevier, 1998.

[11] M. Hennessy and J. Rily. Information flow vs. resource access in the asynchronous pi-calculus (extended abstract). In *ICALP'00*, January 2000.

[12] D. Hoshina, E. Sumii, and A. Yonezawa. A typed process calculus for fine-grained resource access control in distributed computation (full version), 2001. Available from `http://www.yl.is.s.u-tokyo.ac.jp/~hoshina/papers/tacs01full.ps.gz`.

[13] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the pi-calculus. In *POPL'96*. ACM Press, 1996.

[14] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00*. ACM Press, 2000.

[15] R. Milner. The polyadic $\pi$-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science,Department of Computer Science, University of Edinburgh, Octorber 1991.

[16] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structure in Computer Science*, 6(5):409–454, 1996.

[17] B. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *POPL'97*. ACM Press, 1997.

[18] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *POPL'98*. ACM Press, 1998.

[19] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL'99*. ACM Press, 1999.

[20] P. Sewell. Global/local subtyping and capability inference for a distributed $\pi$-calculus. In *ICALP'98*. Springer-Verlag, 1998. LNCS 1433.

[21] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999. LNCS 1686.

[22] N. Yoshida and M. Hennessy. Assigning types to processes. In *LICS'00*. IEEE, 2000.