Towards a Tree of Channels

Xudong Guan [1,2]

INRIA, 2004 Route des Lucioles, 06902 Sophia Antipolis, France

Abstract

This paper presents a generalization of distributed π-calculi to support a hierarchy of locations. We add nested locations on top of a π-calculus core. By unifying channels and locations, we arrive at a computational model which uses mobile agents to pass addresses among immobile and nested locations. We choose a static binding semantics of addresses to enable easy navigation of mobile agents in the location tree. We support dynamic creation of new locations and garbage-collection of empty ones. A sample typed calculus is presented to demonstrate the formalization of type systems and related proof techniques.

1 Introduction

The starting points of our work are distributed π-calculi [1,7,12,17] that model distributed mobile computation by extending π-calculus [9,13] with location constructs. By assigning channels to locations, these calculi can model process migration, resource control, failure, etc. characterizing distributed and mobile computation. However, unlike other mobile computation models [3,15,16,6] using nested location structures, existing distributed π-calculi tend to have a flat layout of locations. A natural question is whether we could have a distributed π-calculus equipped with a hierarchy of locations to model hierarchical administrative domains, hierarchical resource access and security control policy, and hierarchical failure semantics.

In this paper, we study a distributed π-calculus called Tπ having a hierarchy of localities. Roughly speaking, Tπ adds a nestable location construct $\mathtt{a}[P]$ on top of an asynchronous π-calculus core. We generalize the subjects and objects of communication from single names to name strings called *addresses*, pointing to remote locations. Channeled communication in π-calculus is replaced by step-by-step migration following the address and local anonymous communication in the destination.

[1] Supported by EU project 'MIKADO: Mobile Calculi based on Domains', FET-GC IST-2001-32222.

[2] Email: `Xudong.Guan@sophia.inria.fr`

As a basic example, we compare below how a piece of secret can be passed in π, Dπ [7], mobile ambients [3], and Tπ. In the latter three distributed calculi, we try to model as far as possible that the secret is a reference to a fresh sub-location of the sender.

In π, a fresh channel $\mathtt{a}$ can be passed through a shared channel $\mathtt{n}$ and serves as a secret [3]. No physical distribution is modeled.

$$\mathtt{n}(x)p \mid (\nu\,\mathtt{a})(\mathtt{n}\langle\mathtt{a}\rangle \mid q) \longrightarrow (\nu\,\mathtt{a})(p\{x:=\mathtt{a}\} \mid q)$$

In a 2-layered distributed π-calculus like Dπ, a fresh location $\mathtt{a}$ belonging to $\mathtt{l}$ can be passed through a shared channel $\mathtt{n}$ to a remote location $\mathtt{k}$. The sub-location relation is modeled logically.

$$\mathtt{l}[\mathtt{n}(x)p] \mid (\nu\,\mathtt{a})(\mathtt{k}[\mathbf{go}\ \mathtt{l}.\mathtt{n}\langle\mathtt{a}\rangle] \mid \mathtt{a}[q]) \longrightarrow^* (\nu\,\mathtt{a})(\mathtt{l}[p\{x:=\mathtt{a}\}] \mid \mathtt{a}[q])$$

In mobile ambients where migrating ambients form a hierarchy, a route giving access to a fresh sub-ambient $\mathtt{a}$ can be passed as a secret. Please note that the route is hard-coded for the specific receiver $\mathtt{l}$.

$$\begin{aligned}&\mathtt{l}[\mathbf{open}\ \mathtt{n}.(x)p] \mid \mathtt{k}[(\nu\,\mathtt{a})(\mathtt{n}[\mathbf{out}\ \mathtt{k}.\mathbf{in}\ \mathtt{l}.\langle\mathbf{out}\ \mathtt{l}.\mathbf{in}\ \mathtt{k}.\mathbf{in}\ \mathtt{a}\rangle] \mid \mathtt{a}[q])]\\ &\longrightarrow^* (\nu\,\mathtt{a})(\mathtt{l}[p\{x:=\mathbf{out}\ \mathtt{l}.\mathbf{in}\ \mathtt{k}.\mathbf{in}\ \mathtt{a}\}] \mid \mathtt{k}[\mathtt{a}[q]])\end{aligned}$$

In Tπ, we have location hierarchy like ambients. We can send the address of the secret location to some other locations.

$$\mathtt{l}[(x)p] \mid \mathtt{k}[(\nu\,\mathtt{a})(\uparrow.\mathtt{l}\ \langle\mathtt{a}\rangle \mid \mathtt{a}[q])] \longrightarrow^* (\nu\,\mathtt{k}.\mathtt{a})(\mathtt{l}[p\{x:=\uparrow.\mathtt{k}.\mathtt{a}\}] \mid k[\mathtt{a}[q]]) \tag{1}$$

In the above process, an agent $\uparrow.\mathtt{l}\ \langle\mathtt{a}\rangle$ carrying the secret address $\mathtt{a}$ first go up one level ($\uparrow$), then go down into location $\mathtt{l}$, according to the address $\uparrow.\mathtt{l}$. The secret address is translated, when arriving $\mathtt{l}$, to $\uparrow.\mathtt{k}.\mathtt{a}$, the exact address needed by $\mathtt{l}$ to access the remote sub-location. This means that we choose a static semantics for the bindings of addresses. Translation is needed to maintain their bindings during migration. Unlike ambients, immobile location together with static binding semantics of addresses in Tπ bring easy navigation of agents in the location tree. The receiver can either use the received address directly for migration, store it in a mobile agent, or send it elsewhere, with the knowledge that the address will always be maintained during migration and the same destination will always be reached whenever the address is followed.

We study the formalization of such an *address-passing* model among immobile and nested locations. After formalizing the syntax and a few basic definitions in Section 2, we give the address translation function and the reduction semantics in Section 3. Although the calculus is quite reasonable in itself, a small typed version is presented in Section 4 to demonstrate induction proof techniques in the presence of address translation. We study in Section 5 the problem of migrating locations and arbitrary processes. Finally, Section 6 concludes.

[3] Strictly speaking, we need side conditions to avoid name capturing in the examples. As for introduction, we simply assume no name capturing here.

Strings:			Processes:			
s, t	$::= \varepsilon$	empty	P, Q	$::= \mathbf{0}$		empty
	$\mid \mathtt{a}.s$	concatenation		$\mid$	$P \mid P'$	parallel composition
Addresses:				$\mid$	$!\,P$	replication
g, h	$::= s$	string		$\mid$	$\mathtt{a}[P]$	location
	$\mid \uparrow.g$	up one level		$\mid$	$(\nu\, s)P$	restriction, $s \neq \varepsilon$
Values:				$\mid$	$u\,\chi$	mobile agents
u, v	$::= g$	address	Anonymous communication:			
	$\mid x$	variable	χ	$::=$	$\langle \tilde{u} \rangle$	polyadic output
Threads:				$\mid$	$(\tilde{x})p$	polyadic input
p, q	processes without locations					

Fig. 1. Syntax of Tπ

2 Syntax

This section formalizes the syntax of Tπ, together with a few preliminary definitions like α-conversion and structural equivalence.

Given two disjoint countable sets: *labels* ranged over by $\mathtt{a}, \mathtt{b}, ...$ and *variables* ranged over by $x, y, ...$, we define Tπ processes, ranged by $P, Q, ...$, in Fig. 1. Keeping their usual meanings, $\mathbf{0}$ is the null process; $P \mid P'$ denotes the parallel composition of two processes; and $!\,P$ means the composition of infinitely many P's. Location construct $\mathtt{a}[P]$ models nested locations with label $\mathtt{a}$ and process P running inside. *Strings* of labels, ranged over by $s, t, ...$, are used to identify locations, with the empty string ε denoting the current location. Labels distinguish sibling locations only and remote locations with the same label have no relation. As a result, a string is used in the restriction construct, $(\nu\, s)P$, meaning that the location specified by s is not known out of process P. Careful readers may have already noticed this in the introductory example (1), where the binder $(\nu\, \mathtt{a})$ is changed to $(\nu\, \mathtt{k.a})$ after scope extrusion. We require $s \neq \varepsilon$ in the restriction construct since it is unrealistic for a process to restrict its current location.

Strings can only refer to locations within the current node. *Addresses*, ranged over by $g, h, ...$, are used to refer to arbitrary locations in the hierarchy. They are strings with zero or more prefixing up-arrows ($\uparrow$). Like names in π-calculus, addresses are the first class citizen in Tπ. They are the subjects and objects of communication. We call addresses the *structured names* of locations.

Unlike π-calculus, input and output processes in Tπ are mobile. They are written $u\,\chi$ and are called *mobile agents*. A mobile agent migrates according to its subject address u until it reaches the destination. After this, local anonymous communication may happen. Agents can only carry processes without locations called *threads*, ranged over by $p, q, ...$ This will not have any impact on the expressiveness of the language. We will show in Section 5 how arbitrary processes can be transformed into threads for migration.

$$g \bowtie s \triangleq g.s \qquad \uparrow^k \bowtie h \triangleq \uparrow^k.h \qquad g.\mathtt{a} \bowtie \uparrow.h \triangleq g \bowtie h$$

$$\begin{array}{lcllcl} A/_g & \triangleq & \{g.h \mid g.h \in A\} & fa(\mathtt{a}[P]) & \triangleq & \mathtt{a} \bowtie fa(P) \\ A/_{\overline{g}} & \triangleq & \{h \mid h \in A \wedge h \notin A/_g\} & fa((\nu\, s)P) & \triangleq & fa(P)/_{\overline{s}} \end{array}$$

Fig. 2. Free addresses of processes

For simplicity and expressiveness, we choose asynchronous polyadic communication in our syntax: $\langle \tilde{u} \rangle$ stands for a vector of values $u_1, ..., u_k$ waiting to be read, and $(\tilde{x})p$ the receiver that reads a vector of values and substitutes them for the parameters x_i in thread p. Naturally, parameters x_i are assumed to be pair-wise distinct. Unlike Dπ [7] or ${}^e\pi$ [2], values received in Tπ can only be used alone: they can not be used as components to form other addresses.

Conventions: We often omit the trailing ε in strings and addresses, unless ε appears in its own. In writing addresses, $\uparrow^k$ stands for the concatenations of k copies of $\uparrow$. By abuse of notation, we often write $s_1.s_2$, $\uparrow^k.g$, and $g.s$ for strings and addresses when no confusion arises (i.e. any extra ε introduced by these notations does not exist). We also use $g.h$ if the concatenation does not have any $\uparrow$'s appearing after labels. We abbreviate $\varepsilon\, \chi$ as χ. We often omit $\mathbf{0}$ in $(\tilde{x})\mathbf{0}$ and $\mathtt{a}[\mathbf{0}]$. Among the process constructs, composition has the least precedence, i.e. $(\nu\, s)P \mid Q$ stands for $((\nu\, s)P) \mid Q$.

To name sub-processes precisely, we use the notion of *s-indexed process*, inductively defined as follows. A process Q is an *ε-indexed* process, or a *top-level* process, of P, if Q does not appear inside any locations in P. Q is an $\mathtt{a.s}$-*indexed* process of P, if there is an ε-index process $\mathtt{a}[R]$ of P, and Q is an s-indexed process of R. We define s-indexed values similarly. In process $P = (\nu\, s)(P_1 \mid \mathtt{a}[\mathtt{b}[\,!\,P_2 \mid u\,\chi]])$, for example, P_1, P_2, and u are top-level process, $\mathtt{a.b}$-indexed process, and $\mathtt{a.b}$-indexed value, respectively, of P.

To concatenate arbitrary addresses, we define the *chain* of two addresses, $g \bowtie h$, to be the address obtained by removing all the label-dot-up-arrow pairs in their concatenation. We let $g \bowtie A \triangleq \{g \bowtie h \mid h \in A\}$.

We say address $g.h$ *starts with* address g. For a set of addresses A, we use $A/_g$ to denote the subset of A starting with g, and $A/_{\overline{g}}$ the subset not starting with g. Specifically, we let $A/_\varepsilon = A$ and $A/_{\overline{\varepsilon}} = \emptyset$.

Restriction $(\nu\, s)P$ binds all addresses pointing to s in P. The set of *free addresses* of process, $fa(P)$, is defined in Fig. 2 for constructs $\mathtt{a}[P]$ and $(\nu\, s)P$, and has its usual definition for the rests. For example, in process $(\nu\, \mathtt{a})(\mathtt{a}[\langle \uparrow.\mathtt{a} \rangle] \mid \uparrow.\mathtt{b.a}\langle \uparrow.\mathtt{a} \rangle)$, the $\mathtt{a}$-indexed addresses ε and $\uparrow.\mathtt{a}$ are bound (note that $\langle \uparrow.\mathtt{a} \rangle$ is an abbreviation for $\varepsilon\, \langle \uparrow.\mathtt{a} \rangle$), but the top-level addresses $\uparrow.\mathtt{b.a}$ and $\uparrow.\mathtt{a}$ are free, even though they happen to contain the same label $\mathtt{a}$. As in π-calculus, $u\,(\tilde{x})P$ binds $\tilde{x}$ in P and the set of free variables is defined accordingly. We will not state explicitly the corresponding definitions for threads, since they are just special cases of those of processes.

$$
\begin{array}{rcll}
\mathbf{rfl}_t(\mathtt{c}[P], s, \mathtt{a}, \mathtt{b}) & \stackrel{\triangle}{=} & \mathtt{b}[\mathbf{rfl}_{t.\mathtt{c}}(P, s, \mathtt{a}, \mathtt{b})] & \text{if } t = s \text{ and } \mathtt{a} = \mathtt{c} \\
 & & \mathtt{c}[\mathbf{rfl}_{t.\mathtt{c}}(P, s, \mathtt{a}, \mathtt{b})] & \text{otherwise} \\
\mathbf{rfl}_t((\nu\, s')P, s, \mathtt{a}, \mathtt{b}) & \stackrel{\triangle}{=} & (\nu\, s'_1.\mathtt{b}.s'_2)\mathbf{rfl}_t(P, s, \mathtt{a}, \mathtt{b}) & \text{if } s' = s'_1.\mathtt{a}.s'_2 \text{ and } s \bowtie s'_1 = t \\
 & & (\nu\, s')\mathbf{rfl}_t(P, s, \mathtt{a}, \mathtt{b}) & \text{otherwise} \\
\mathbf{rfl}_t(g, s, \mathtt{a}, \mathtt{b}) & \stackrel{\triangle}{=} & g'.\mathtt{b}.s' & \text{if } g = g'.\mathtt{a}.s' \text{ and } t \bowtie g' = s \\
 & & g & \text{otherwise}
\end{array}
$$

Fig. 3. Renaming of free locations

Symm	$P \equiv P$	Par	$P \equiv Q \Longrightarrow P \mid R \equiv Q \mid R$
Refl	$P \equiv Q \Longrightarrow Q \equiv P$	Res	$P \equiv Q \Longrightarrow (\nu\, s)P \equiv (\nu\, s)Q$
Tran	$P \equiv Q, Q \equiv R \Longrightarrow P \equiv R$	Loc	$P \equiv Q \Longrightarrow \mathtt{a}[P] \equiv \mathtt{a}[Q]$
Split	$\mathtt{a}[P \mid Q] \equiv \mathtt{a}[P] \mid \mathtt{a}[Q]$	Par-comm	$P \mid Q \equiv Q \mid P$
Garb	$\mathtt{a}[\mathbf{0}] \equiv \mathbf{0}$	Par-assoc	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
Alpha	$P \equiv_\alpha Q \Longrightarrow P \equiv Q$	Rep-par	$!\, P \equiv P \mid !\, P$
Res-par	$P \mid (\nu\, s)Q \equiv (\nu\, s)(P \mid Q)$,	if $fa(P)/_s = \emptyset$	
Res-loc	$\mathtt{a}[(\nu\, s)P] \equiv (\nu\, \mathtt{a}.s)\mathtt{a}[P]$,	if $fa(P)/_{\uparrow.\mathtt{a}.s} = \emptyset$	

Fig. 4. Structural equivalence of processes

We identify processes up to renaming of bound variables and locations. Renaming of bound variables has its normal definition. Let's consider now renaming of bound locations using example $(\nu\, \mathtt{a})(\nu\, \mathtt{a.c})\mathtt{a}[\mathtt{c}[\langle\uparrow.\uparrow.\mathtt{a}\rangle]]$. To rename location $\mathtt{a}$ to $\mathtt{b}$, we need to change the binder itself, the binder $(\nu\, \mathtt{a.c})$, the label $\mathtt{a}$, and the address $\uparrow.\uparrow.\mathtt{a}$, resulting process $(\nu\, \mathtt{b})(\nu\, \mathtt{b.c})\mathtt{b}[\mathtt{c}[\langle\uparrow.\uparrow.\mathtt{b}\rangle]]$. In Fig. 3, we define *renaming of free locations* for sub-processes and values within the scope of the binder. In $\mathbf{rfl}_t(X, s, \mathtt{a}, \mathtt{b})$, X is a t-indexed process or value, the free location to be renamed is $s.\mathtt{a}$, and $\mathtt{b}$ is the fresh label. As usual, we omit trivial and analogous ones including those for threads.

Now, we define α-conversion as (where $\mathtt{b}$ is chosen fresh):

$$(\nu\, s.\mathtt{a})P \equiv_\alpha (\nu\, s.\mathtt{b})\mathbf{rfl}_\varepsilon(P, s, \mathtt{a}, \mathtt{b})$$

Structural equivalence: We define structural equivalence between processes in Fig. 4, most of which are quite standard. Among those rules, two interesting ones are Res-par and Res-loc. In rule Res-par, condition $fa(P)/s = \emptyset$ avoids capture of free addresses in P by $(\nu\, s)$. The condition can always be satisfied by renaming of location s in Q. Rule Res-loc says that a restriction string, when extruding out of a location, needs to be prefixed with the label of that location. Moreover, condition $fa(P)/_{\uparrow.\mathtt{a}.s} = \emptyset$ prevents free addresses like $\uparrow.\mathtt{a}.s$ in P from becoming bound after scope extrusion. Consider for example process $P = \mathtt{a}[\mathtt{b}[\ ] \mid (\nu\, \mathtt{b})\uparrow.\mathtt{a}.\mathtt{b}\langle\mathtt{b}\rangle]$. By definition in Fig. 2, $fa(P) = \{\mathtt{a.b}\}$. Address $\uparrow.\mathtt{a.b}$ is free and refers to the free location $\mathtt{a.b}$ in P. Without the side condition, we would have $P \equiv (\nu\, \mathtt{a.b})\mathtt{a}[\mathtt{b}[\ ] \mid \uparrow.\mathtt{a.b}\langle\mathtt{b}\rangle]$ and the free address would become captured. Rules Split is commonly adopted in

$$
\begin{array}{rclrcll}
\mathtt{a} \oplus g & \triangleq & \mathtt{a} \bowtie g & {}^s h \oplus_A (\nu\, s')p & \triangleq & (\nu\, s')\ {}^s h \oplus_{A\cup\{s'\}} p & \\
{}^{\mathtt{a}}\!\uparrow \oplus g & \triangleq & t \quad \text{if } g = \mathtt{a}.t & {}^s h \oplus_A u & \triangleq & {}^s h \oplus g & \text{if } u = g \notin A \\
 & & \uparrow \bowtie g \quad \text{otherwise} & & & u & \text{otherwise}
\end{array}
$$

$$
{}^s h \oplus g \triangleq {}^{\mathtt{a_k}}\!\uparrow \oplus (\ldots {}^{\mathtt{a_1}}\!\uparrow \oplus (\mathtt{b_1} \oplus \ldots \oplus \mathtt{b_i} \oplus g)\ldots), \text{ if } s = s_1.\mathtt{a_1}\ldots\mathtt{a_k} \wedge h = \uparrow^k.\mathtt{b_1}\ldots\mathtt{b_i}
$$

Fig. 5. Address translation

distributed π-calculi for a concise reduction semantics.

For structural equivalence, we have:

Lemma 2.1 *If $P \equiv Q$, then $fa(P) = fa(Q)$.*

Proof. The proof is easy by the definitions of $fa(P)$ and $\equiv$. □

In this section, we present the syntax of Tπ, and define free addresses, α-conversion, and structural equivalence. We are now ready to present the operational semantics of Tπ.

3 Operational Semantics

We give the operational semantics of Tπ in this section using a simple binary reduction relation. We first formalize the address translation function during migration. We then present the reduction rules and illustrate with our introductory example (1).

To maintain the static binding semantics of addresses in Tπ, we need to update all the free addresses in migrating agents. The definition of address translation is summarized in Fig. 5. Address g becomes $\mathtt{a} \oplus g$ after it comes out of location $\mathtt{a}$, it becomes ${}^{\mathtt{a}}\!\uparrow \oplus g$ after it goes down to location $\mathtt{a}$. They are defined such that an address always points to the same location during migration. Address bindings should recover to their original forms after agents migrate back to their original locations. This is indeed the case since it is easy to check that $\mathtt{a} \oplus ({}^{\mathtt{a}}\!\uparrow \oplus g) = {}^{\mathtt{a}}\!\uparrow \oplus (\mathtt{a} \oplus g) = g$. We also generalize them and define *batch translation*, ${}^s h \oplus g$, that translates addresses after a series of migrations denoted by ${}^s h$.

When translating mobile agents, we recursively update all the free addresses in them. We keep restricted addresses untouched so that mobile agents can create fresh locations dynamically. Let X stands for p or χ, the result of the translation, ${}^s h \oplus_A X$, is defined in Fig. 5. The subscript A is the set of restricted addresses remembered during the translation. We only include here definitions for translating values and restrictions, the other constructs are analogous. By definition, we have ${}^s h \oplus_\emptyset g = {}^s h \oplus g$ and ${}^s \mathtt{a_1}\ldots\mathtt{a_k} \oplus g = {}^\varepsilon \mathtt{a_1}\ldots\mathtt{a_k} \oplus g = \mathtt{a_1} \oplus \ldots \oplus \mathtt{a_k} \oplus g$. For simplicity, we often abbreviate ${}^s h \oplus_\emptyset X$ to ${}^s h \oplus X$, and ${}^s t \oplus X$ to $t \oplus X$.

The reduction rules of Tπ are summarized in Fig. 6. The two migration rules, Up and Dn, are defined using address translation. They enable one step

$$
\begin{array}{llcl}
\textsc{Up} & \mathtt{a}[\uparrow.g\,\chi] & \longrightarrow & g\,(\mathtt{a}\oplus\chi) \\
\textsc{Dn} & \mathtt{a}.g\,\chi & \longrightarrow & \mathtt{a}[g\;\;({}^{\mathtt{a}}\!\uparrow\oplus\chi)] \\
\textsc{Comm} & (\tilde{x})p \mid \langle\tilde{u}\rangle & \longrightarrow & p\{\tilde{x}:=\tilde{u}\}
\end{array}
\qquad
\begin{array}{lrcl}
\textrm{R-ctx} & P\longrightarrow Q & \Longrightarrow & P\mid R\longrightarrow Q\mid R \\
 & P\longrightarrow Q & \Longrightarrow & (\nu\, s)P\longrightarrow(\nu\, s)Q \\
 & P\longrightarrow Q & \Longrightarrow & \mathtt{a}[P]\longrightarrow\mathtt{a}[Q]
\end{array}
$$

$$
\textrm{R-struct}\qquad P\equiv P',\, P'\longrightarrow P'',\, P''\equiv P''' \quad\Longrightarrow\quad P\longrightarrow P'''
$$

Fig. 6. Reduction rules of Tπ

migrations of agents within the hierarchy. Rule Comm enables anonymous communication of values between co-located processes. Since no explicit locations are presented in threads, *substitution* of free variables in threads with values, written $p\{\tilde{x}:=\tilde{u}\}$, is defined as usual [4], with implicit use of α-conversion to avoid capture. We have four other standard rules that enable reductions to happen inside unguarded contexts, and enable structural re-arrangements before and after reductions.

We illustrate the use of address translation and reduction rules with our introductory example (1), restated below for easy cross referencing.

$$\mathtt{l}[(x)p] \mid \mathtt{k}[(\nu\,\mathtt{a})(\uparrow.\mathtt{l}\,\langle\mathtt{a}\rangle \mid \mathtt{a}[q])]$$

In order for the mobile agent $\uparrow.\mathtt{l}\,\langle\mathtt{a}\rangle$ inside $\mathtt{k}$ to move up, we need first to extrude the binder $(\nu\,\mathtt{a})$ out of $\mathtt{k}$ with Res-loc. Suppose $fa(\mathtt{a}[q])/_{\uparrow.\mathtt{k}.\mathtt{a}} = \emptyset$ (otherwise rename bound location $\mathtt{a}$), we have $fa(\uparrow.\mathtt{l}\,\langle\mathtt{a}\rangle \mid \mathtt{a}[q])/_{\uparrow.\mathtt{k}.\mathtt{a}} = (\{\uparrow.\mathtt{l},\,\mathtt{a}\}\cup fa(\mathtt{a}[q]))/_{\uparrow.\mathtt{k}.\mathtt{a}} = \emptyset$.

$$\equiv \mathtt{l}[(x)p] \mid (\nu\,\mathtt{k}.\mathtt{a})\mathtt{k}[\uparrow.\mathtt{l}\,\langle\mathtt{a}\rangle \mid \mathtt{a}[q]]$$

By separating $\mathtt{k}$ using Split, we have:

$$\equiv \mathtt{l}[(x)p] \mid (\nu\,\mathtt{k}.\mathtt{a})(\mathtt{k}[\uparrow.\mathtt{l}\,\langle\mathtt{a}\rangle] \mid \mathtt{k}[\mathtt{a}[q]])$$

Now we can have an upward movement using rule Up and R-ctx. The address $\mathtt{a}$ becomes $\mathtt{k}\oplus\mathtt{a} = \mathtt{k}.\mathtt{a}$ after address translation.

$$\longrightarrow \mathtt{l}[(x)p] \mid (\nu\,\mathtt{k}.\mathtt{a})(\mathtt{l}\,\langle\mathtt{k}.\mathtt{a}\rangle \mid \mathtt{k}[\mathtt{a}[q]])$$

A downward movement into location $\mathtt{l}$ can follow immediately, turning $\mathtt{k}.\mathtt{a}$ to ${}^{l}\!\uparrow\oplus\mathtt{k}.\mathtt{a} = \uparrow.\mathtt{k}.\mathtt{a}$ this time.

$$\longrightarrow \mathtt{l}[(x)p] \mid (\nu\,\mathtt{k}.\mathtt{a})(\mathtt{l}[\langle\uparrow.\mathtt{k}.\mathtt{a}\rangle] \mid \mathtt{k}[\mathtt{a}[q]])$$

Suppose that $fa(\mathtt{a}[(x)p])/_{\mathtt{k}.\mathtt{a}} = \emptyset$ (otherwise rename bound location $\mathtt{k}.\mathtt{a}$), we can use the scope extrusion rule Res-par:

$$\equiv (\nu\,\mathtt{k}.\mathtt{a})(\mathtt{l}[(x)p] \mid \mathtt{l}[\langle\uparrow.\mathtt{k}.\mathtt{a}\rangle] \mid \mathtt{k}[\mathtt{a}[q]])$$

We use rule Split to join the two processes at location $\mathtt{l}$:

$$\equiv (\nu\,\mathtt{k}.\mathtt{a})(\mathtt{l}[(x)p \mid \langle\uparrow.\mathtt{k}.\mathtt{a}\rangle] \mid \mathtt{k}[\mathtt{a}[q]])$$

Finally, local communication happens, producing the following process that we use in (1).

$$\longrightarrow (\nu\,\mathtt{k}.\mathtt{a})(\mathtt{l}[p\{x:=\uparrow.\mathtt{k}.\mathtt{a}\}] \mid \mathtt{k}[\mathtt{a}[q]])$$

[4] Substituting free variables in arbitrary processes will be a little complex, requiring a rule like $\mathtt{a}[P]\{x:=\alpha\} \stackrel{\triangle}{=} \mathtt{a}[P\{x:={}^{\mathtt{a}}\!\uparrow\oplus\alpha\}]$ for the location construct.

We now state an important property of our reduction semantics, i.e., reduction will never introduce new free addresses.

Lemma 3.1 *If $P \longrightarrow Q$, then $fa(Q) \subseteq fa(P)$.*

Proof (Sketch) For rule Comm, it is easy to check since substitution will not introduce free addresses. For the migration rules, we can check the result by the definition of address translation. The other rules can be proved by induction, with the help of Lemma 2.1 for the rule R-struct. $\Box$

To this point, we have arrived at a reasonable formalization of our address-passing model with nested locations. We have a tree of channels whose addresses can be passed around. Agents migrate according to addresses, and communication can happen at any level in the tree. In the following sections, we study a simple type version eliminating communication errors, and we discuss the problem of migrating arbitrary processes.

4 Typed Calculus

In this section, we study a typed version of Tπ assigning Milner's sorts [8] to locations to eliminate communication errors for well-typed processes. The type system is standard and simple. We focus here on the presentation of type systems and the related proof methods in the presence of structured names and address relocation.

Value types are ranged over by $\mathrm{U}, \mathrm{V}, ...$, and the typed version is obtained in the standard way by adding type annotations for location and variable binders in the syntax. All values in Tπ have address type of the form $\mathtt{loc}(\tilde{\mathrm{V}})$, where $\tilde{\mathrm{V}}$ is the vector of value types exchanged in the location.

A type environment Γ is a collection of assignments of values to types of the form $u{:}\mathrm{U}$. A *well-formed* environment Γ doesn't contain multiple entries for a same value. We identify environments up to reordering of assignments. We extend address translation "${}^{s}h \oplus_{A} \cdot$" to type environments. Please note that well-formedness is not preserved by address translation. Consider for example $\Gamma = \mathtt{b}:\mathrm{U}_1,\ \uparrow.\mathtt{a}.\mathtt{b}:\mathrm{U}_2$, the environment $\mathtt{a} \oplus \Gamma$ will have multiple entries for address $\mathtt{a}.\mathtt{b}$. In case $\mathrm{U}_1 = \mathrm{U}_2$, we assume the duplicate removed. Otherwise, $\mathtt{a} \oplus \Gamma$ is not well-formed.

The typing rules in Fig. 7 are quite standard. The most interesting one due to structured names is T-loc. To get the right type environment for process $\mathtt{a}[P]$ from $\Gamma \vdash P$, we need to update the free addresses in Γ by prefixing them with label $\mathtt{a}$. We require $\mathtt{a} \oplus \Gamma$ to be also well-formed. In rule T-res, the side condition ensures that the types of all the sub-locations of the restricted location have already been declared. Consider for example process $(\nu\,\mathtt{a}{:}\mathrm{U})\mathtt{a}[\mathtt{b}[\langle\tilde{v}\rangle]]$. This process is untypable since the type of location $\mathtt{a}.\mathtt{b}$ is not known. One should write instead $(\nu\,\mathtt{a}:\mathrm{U})(\nu\,\mathtt{a}.\mathtt{b}:\mathrm{V})\mathtt{a}[\mathtt{b}[\langle\tilde{v}\rangle]]$ for some proper type V of location $\mathtt{a}.\mathtt{b}$.

Well-formed environments:

$$\text{T-EMPTY } \emptyset \vdash \diamond \qquad \text{T-INTRO } \frac{\Gamma \vdash \diamond \quad u \notin dom(\Gamma)}{\Gamma, u{:}\mathrm{U} \vdash \diamond} \qquad \text{T-VAL } \Gamma, u{:}\mathrm{U} \vdash u{:}\mathrm{U}$$

Well-typed processes:

$$\text{T-NIL } \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}} \qquad \text{T-PAR } \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \text{T-REP } \frac{\Gamma \vdash P}{\Gamma \vdash !\,P}$$

$$\text{T-RES } \frac{\Gamma, s{:}\mathrm{U} \vdash P \quad fa(P)/_s \subseteq \{s\}}{\Gamma \vdash (\nu\, s{:}\mathrm{U})P} \qquad \text{T-LOC } \frac{\Gamma \vdash P \quad \mathtt{a} \oplus \Gamma \vdash \diamond}{\mathtt{a} \oplus \Gamma \vdash \mathtt{a}[P]}$$

$$\text{T-IN } \frac{\Gamma \vdash u{:}\mathtt{loc}(\tilde{\mathrm{U}}) \quad \Gamma, \tilde{x}{:}\tilde{\mathrm{U}} \vdash p}{\Gamma \vdash u\,(\tilde{x}{:}\tilde{\mathrm{U}})p} \qquad \text{T-OUT } \frac{\Gamma \vdash u{:}\mathtt{loc}(\tilde{\mathrm{U}}), \tilde{v}{:}\tilde{\mathrm{U}}}{\Gamma \vdash u\,\langle \tilde{v} \rangle}$$

Fig. 7. Typing rules

$$\begin{array}{rcll} {}^{\mathtt{a}}\!\uparrow \bullet \emptyset & \triangleq & \emptyset & \\ {}^{\mathtt{a}}\!\uparrow \bullet (\Gamma \uplus \{u : \mathrm{U}\}) & \triangleq & ({}^{\mathtt{a}}\!\uparrow \bullet \Gamma) \uplus \{s{:}\mathrm{U},\ \uparrow.\mathtt{a}.s{:}\mathrm{U}\} & u = \mathtt{a}.s \\ & & ({}^{\mathtt{a}}\!\uparrow \bullet \Gamma) \uplus \{({}^{\mathtt{a}}\!\uparrow \oplus u){:}\mathrm{U}\} & \text{otherwise} \end{array}$$

Fig. 8. Back-tracing an environment

The main interest in this section is the proof for the subject reduction theorem in the presence of address translation during reductions. First, to use inductive proof, we need to find the right environment for P from $\Gamma \vdash \mathtt{a}[P]$. It is the *back-trace* of Γ, ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma$, defined in Fig. 8. We use $\uplus$ for the union of (disjoint) type environments. It is easy to check that ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma$ is always valid if Γ valid.

Lemma 4.1 *The followings hold:*

(1) $\mathtt{a} \oplus ({}^{\mathtt{a}}\!\uparrow \oplus \Gamma) = \Gamma$*;*

(2) $\mathtt{a} \oplus ({}^{\mathtt{a}}\!\uparrow \bullet \Gamma) = \Gamma$*;*

(3) If $\Gamma \vdash \diamond$*, then* ${}^{\mathtt{a}}\!\uparrow \oplus \Gamma \vdash \diamond$*;*

(4) If $\Gamma \vdash \diamond$*, then* ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma \vdash \diamond$.

Proof. (1)(2) are easily checked by definition. For (3), we may check that the function " ${}^{\mathtt{a}}\!\uparrow \oplus \cdot$" on addresses is injective, so the result is well-formed if the source is well-formed. For (4), although one assignment in Γ may become two in ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma$, there will still be no duplicates by definition if the original environment is well-formed. □

The following result justifies our definition of back-trace, and enables inductive reasoning with $\Gamma \vdash P$.

Lemma 4.2 *If* $\Gamma \vdash \mathtt{a}[P]$*, then* ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma \vdash P$.

Proof. The result $\Gamma \vdash \mathtt{a}[P]$ can only be obtained by T-LOC with $\Gamma' \vdash P$ and $\Gamma = \mathtt{a} \oplus \Gamma'$. We can show that ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma$ always maps a value to the same type that Γ' maps to, that is, ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma \vdash P$ is a weaken result of $\Gamma' \vdash P$. □

The following one is the stand point for proving subject reduction of migrations.

Lemma 4.3 *If* $\Gamma \vdash p$ *and* ${}^s g \oplus_A \Gamma \vdash \diamond$, *then* ${}^s g \oplus_A \Gamma \vdash\ {}^s g \oplus_A p$.

Proof. Each value u in Γ used to prove $\Gamma \vdash p$ can be used in the same way but as ${}^s g \oplus_A u$ in the proof of ${}^s g \oplus_A \Gamma \vdash\ {}^s g \oplus_A p$. □

Now we may prove the subject reduction theorem, which guarantees that communication errors never happen in well-typed processes.

Theorem 4.4 (Subject reduction) *If* $\Gamma \vdash P$ *and* $P \longrightarrow Q$, *then* $\Gamma \vdash Q$.

Proof. By induction on the derivation of $\Gamma \vdash P$. We only show the case of DN:
Case DN**):** In this case we have $\Gamma \vdash \mathtt{a}[\uparrow.g\,\chi]$. By Lemma 4.2 we have ${}^{\mathtt{a}}\!\uparrow \bullet \Gamma \vdash \uparrow.g\,\chi$. By Lemma 4.1.(2), we know $\Gamma = \mathtt{a} \oplus ({}^{\mathtt{a}}\!\uparrow \bullet \Gamma)$. By Lemma 4.3 we have $\Gamma \vdash \mathtt{a} \oplus (\uparrow.g\,\chi)$. That is, $\Gamma \vdash g\,\mathtt{a} \oplus \chi$. □

We end this section with an example. Let U be some type, we write below a simplified case of example (1) (where $p = !\,\langle x\rangle$ and $q = \mathbf{0}$) using type annotations.

$$\mathtt{l}[(x{:}\mathrm{U})\,!\,\langle x\rangle] \mid \mathtt{k}[(\nu\,\mathtt{a}{:}\mathrm{U})\uparrow.\mathtt{l}\,\langle \mathtt{a}\rangle]$$

We show how to derive the following type judgment using our type system:

$$\mathtt{l}{:}\mathtt{loc}(\mathrm{U}) \vdash \mathtt{l}[(x{:}\mathrm{U})\,!\,\langle x\rangle] \mid \mathtt{k}[(\nu\,\mathtt{a}{:}\mathrm{U})\uparrow.\mathtt{l}\,\langle \mathtt{a}\rangle] \tag{2}$$

We first detail the derivation of $\mathtt{l}{:}\mathtt{loc}(\mathrm{U}) \vdash \mathtt{l}[(x{:}\mathrm{U})\,!\,\langle x\rangle]$ below:

(a)	$\emptyset$	$\vdash \diamond$	T-EMPTY
(b)	$x{:}\mathrm{U}$	$\vdash \diamond$	(a)+T-INTRO
(c)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U})$	$\vdash \diamond$	(b)+T-INTRO
(d)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U})$	$\vdash x{:}\mathrm{U}$	(c)+T-VAL
(e)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U})$	$\vdash \varepsilon{:}\mathtt{loc}(\mathrm{U})$	(c)+T-VAL
(f)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U})$	$\vdash \langle x\rangle$	(d)(e)+T-OUT
(g)	$x{:}\mathrm{U}, \varepsilon{:}\mathtt{loc}(\mathrm{U})$	$\vdash !\,\langle x\rangle$	(f)+T-REP
(h)	$\varepsilon{:}\mathtt{loc}(\mathrm{U})$	$\vdash \diamond$	(a)+T-INTRO
(i)	$\varepsilon{:}\mathtt{loc}(\mathrm{U})$	$\vdash \varepsilon{:}\mathtt{loc}(\mathrm{U})$	(h)+T-VAL
(j)	$\varepsilon{:}\mathtt{loc}(\mathrm{U})$	$\vdash (x{:}\mathrm{U})\,!\,\langle x\rangle$	(g)(i)+T-IN
(k)	$\mathtt{l}{:}\mathtt{loc}(\mathrm{U})$	$\vdash \mathtt{l}[(x{:}\mathrm{U})\,!\,\langle x\rangle]$	(j)+T-LOC

The last step using T-LOC is the most interesting. Notice how the environment $\varepsilon : \mathtt{loc}(\mathrm{U})$ becomes $\mathtt{l} : \mathtt{loc}(\mathrm{U})$ by address translation on environments. The derivation of $\mathtt{l}{:}\mathtt{loc}(\mathrm{U}) \vdash \mathtt{k}[(\nu\,\mathtt{a}{:}\mathrm{U})\uparrow.\mathtt{l}\,\langle \mathtt{a}\rangle]$ is similar. Note that we need to check $fa(\uparrow.\mathtt{l}\,\langle \mathtt{a}\rangle)/_{\mathtt{a}} \subseteq \{\mathtt{a}\}$ when using rule T-RES. Judgment (2) is the combination of these two results using T-PAR.

This section presents an example of how to develop type system in the presence of structured names. Although the type system is not sought for resolving any particular problem of the language, rules like T-LOC and T-RES require a few careful thought. Moreover, results like Lemma 4.2 and 4.3

demonstrate how to enable induction on judgment and reasoning with address translation. More powerful type systems expressing e.g. I/O sub-typing [10] can be developed further with similar proof techniques.

5 Migrating Arbitrary Processes

In this section, we give examples illustrating the expressiveness of Tπ, especially the migration of arbitrary processes, which seems to be limited by the syntax. We first rewrite the well-known reference cell example in Tπ. We show how a cell can be accessed uniformly by remote nodes at different locations. As a cell uses a private location to keep the value, it can not be migrated directly as threads. We provide a way of converting processes to threads, and we show how to enable mobile agents creating cells dynamically in any remote locations.

All examples in this section are written in the untyped version, since annotating types in processes will make them not as readable.

Defining a cell: We rewrite the cell example from [7] in Tπ. We define a cell process that can keep one piece of value. A user may invoke on the two methods provided, put and get, to access the cell.

$$\begin{aligned} Cell(u) \triangleq\ & (\nu\, \mathtt{a})\ (\mathtt{a}[\langle u\rangle] \\ & \mid !\,\mathtt{put}\,(y,z)\ \mathtt{a}\,(x)(a\,\langle y\rangle \mid z\,\langle\rangle) \\ & \mid !\,\mathtt{get}\quad (z)\ \mathtt{a}\,(x)(\mathtt{a}\,\langle x\rangle \mid z\,\langle x\rangle)\ ) \end{aligned}$$

The value is kept inside a private location a in the cell. Two public methods are provided as the interface for the cell. Knowing the two methods, we may define a user that stores a value v in the cell, reads that value, and sends it to a printer.

$$\begin{aligned} User(g,h) \triangleq\ & (\nu\, \mathtt{ack})(\nu\, \mathtt{ret}) \\ & \quad (g\,\langle v, \mathtt{ack}\rangle \\ & \quad \mid \mathtt{ack}\,()(h\,\langle \mathtt{ret}\rangle \mid \mathtt{ret}\,(x)\ \mathtt{print}\langle x\rangle) \end{aligned}$$

Now we may have the following interactions.

$$Cell(u) \mid User(\mathtt{put},\mathtt{get}) \longrightarrow^{*} Cell(v) \mid \mathtt{print}\langle v\rangle \tag{3}$$

One advantage of Tπ is that programmers need not care much about the location where the code will be deployed. This frees the programmers from writing dedicated routing code in designing applications, thus supports better code reuse. As an example, the user and the cell can reside at any node in the network. They can interact correctly as long as the user knows the right addresses of the two methods. Readers may check the following reductions.

$$\mathtt{svr}[Cell(u)] \mid User(\mathtt{svr.put},\mathtt{svr.get}) \longrightarrow^{*} \mathtt{svr}[Cell(v)] \mid \mathtt{print}\langle v\rangle$$

$$\begin{aligned} & \mathtt{svr}[Cell(u)] \mid \mathtt{usr}[User(\uparrow.\mathtt{svr.put},\uparrow.\mathtt{svr.get})] \\ & \qquad\qquad \longrightarrow^{*} \mathtt{svr}[Cell(v)] \mid \mathtt{usr}[\mathtt{print}\langle v\rangle] \end{aligned}$$

Migrating locations: One might argue that it is a severe limit that agents in Tπ can only carry threads, not arbitrary processes having locations inside. In this example, we will show how to create dynamic locations in Tπ.

First we define the following abbreviation:

$$\mathbf{go}\ g.X \quad \stackrel{\triangle}{=} \quad g\,\langle\rangle \mid g\,()X$$

We can use this abbreviation to migrate arbitrary threads:

$$\mathbf{go}\ \mathtt{a}.p \ \longrightarrow^{*}\ \mathtt{a}[{}^{\mathtt{a}}\!\uparrow \oplus p]$$

Now we show how to create dynamic locations by carrying them in mobile agents. For example, instead of writing

$$\langle g\rangle \mid \langle h\rangle \mid (x)\mathbf{go}\ x.(\nu\,\mathtt{a})(\mathtt{a}[\,!\,\langle\uparrow.\mathtt{b}\rangle] \mid \mathtt{c}\,\langle\mathtt{a}\rangle),$$

which is not a valid Tπ process, we can write

$$\langle g\rangle \mid \langle h\rangle \mid (x)\mathbf{go}\ x.(\nu\,\mathtt{a})(\,!\,\mathtt{a}\,\langle\mathtt{b}\rangle \mid \mathtt{c}\,\langle\mathtt{a}\rangle),$$

which is a valid process and the destination of location $\mathtt{a}$ can only be determined at runtime. We may check that the two processes $(\nu\,\mathtt{a})(\mathtt{a}[\,!\,\langle\uparrow.\mathtt{b}\rangle] \mid \mathtt{c}\,\langle\mathtt{a}\rangle)$ and $(\nu\,\mathtt{a})(\,!\,\mathtt{a}\,\langle\mathtt{b}\rangle \mid \mathtt{c}\,\langle\mathtt{a}\rangle)$ have the same behavior. The former has an explicit location construct, while the latter hides the explicit location in its addresses.

Similarly, we can rewrite the cell process as a thread:

$$\begin{array}{lll} \mathit{Cell}'(u) \stackrel{\triangle}{=} (\nu\,\mathtt{a}) & (& \mathtt{a}\,\langle\mathtt{a}\oplus u\rangle \\ & \mid & !\,\mathtt{put}\,(y,z)\ \mathtt{a}\,(x)(a\,\langle y\rangle \mid z\,\langle\rangle) \\ & \mid & !\,\mathtt{get}\quad (z)\ \mathtt{a}\,(x)(\mathtt{a}\,\langle x\rangle \mid z\,\langle x\rangle)\ ) \end{array}$$

Following Dπ, we may now write a cell server able to create cells anywhere according to client requests.

$$\begin{array}{lll} \mathit{Server} & \stackrel{\triangle}{=} & \mathtt{req}[\,!\,(\mathit{dest},\mathit{ret},x)\ \mathbf{go}\ \mathit{dest}\ .\ (\nu\,\mathtt{put})(\nu\,\mathtt{get}) \\ & & \qquad\qquad\qquad\qquad (\mathit{ret}\langle\mathtt{put},\mathtt{get}\rangle \mid \mathit{Cell}'(x)\ )] \\ \mathit{Client}(g) & \stackrel{\triangle}{=} & (\nu\,\mathtt{a})\ (\ g\,\langle\mathtt{a},\varepsilon,u\rangle \mid (x,y)\ \mathit{User}(x,y)) \end{array}$$

Given a destination *dest*, a return address *ret*, and an initial value x, the cell server creates dynamically a cell at *dest* with initial value x, and sends the addresses of the two methods to *ret*. A typical configuration like the following works.

$$\begin{array}{ll} & \mathtt{svr}[\mathit{Server}] \mid \mathtt{usr}[\mathit{Client}(\uparrow.\mathtt{svr.req})] \\ \longrightarrow & \mathtt{svr}[\mathit{Server}] \mid \mathtt{usr}\ [\ (\nu\,\mathtt{a})(\nu\,\mathtt{a.put})(\nu\,\mathtt{a.get}) \\ & \qquad\qquad\qquad (\ \mathtt{a}[\mathit{Cell}'({}^{\mathtt{a}}\!\uparrow \oplus u)] \mid \mathit{User}(\mathtt{a.put},\mathtt{a.get})\ )\ ] \end{array}$$

The user now has the sole access to the newly created cell at location $\mathtt{a}$.

Migrating arbitrary processes: To go a step further, we now show how to transform arbitrary processes to threads, so as to support migration of arbitrary processes in Tπ. We define in Fig. 9 the transformation of s-indexed processes to threads: $\langle\!\langle P\rangle\!\rangle_s$.

For any process, we can show that P and $\langle\!\langle P\rangle\!\rangle_\varepsilon$ have the same external

$$
\begin{array}{rclcrcl}
\langle\!\langle \mathbf{0} \rangle\!\rangle_s & \triangleq & \mathbf{0} & \qquad & \langle\!\langle \mathsf{a}[P] \rangle\!\rangle_s & \triangleq & \langle\!\langle P \rangle\!\rangle_{s.\mathsf{a}} \\
\langle\!\langle P \mid Q \rangle\!\rangle_s & \triangleq & \langle\!\langle P \rangle\!\rangle_s \mid \langle\!\langle Q \rangle\!\rangle_s & \qquad & \langle\!\langle (\nu\, t) P \rangle\!\rangle_s & \triangleq & (\nu\, s.t) \langle\!\langle P \rangle\!\rangle_s \\
\langle\!\langle\, !\, P \rangle\!\rangle_s & \triangleq & !\, \langle\!\langle P \rangle\!\rangle_s & \qquad & \langle\!\langle u\, \chi \rangle\!\rangle_s & \triangleq & (s \oplus u)\, (s \oplus \chi)
\end{array}
$$

Fig. 9. Transforming s-indexed processes to threads

behavior [5], while the latter is a thread without any explicit location. As a result, we can write mobile agents like $u\,(x)P$ as a short hand for $u\,(x)\langle\!\langle P \rangle\!\rangle_\varepsilon$. We can now justify our claim in Section 2, that the limitation in the syntax doesn't have any impact on expressiveness. It only makes a simpler definition for address translation and substitution.

One may wonder on the other hand why we need explicit location construct in the syntax, since every process has a corresponding thread version with the same behavior. We now make the following explanations. As a mobile and distributed computation model, Tπ needs locations for modeling network and computational nodes. Although virtual locations within a same computational node can be eliminated, physical locations representing these nodes can not be. They are essential to express distribution, migration, and security control. Without the location construct, we would go back to a single machine model. The resulting calculus would become to some extent just an asynchronous π-calculus with the set of strings as its channels (not ${}^e\pi$, since addresses must be used as a whole).

Comparing Tπ with π-calculus and $lsd\pi$ [12]: By thinking π-calculus channels as a flat layer of locations, we may take (asynchronous) π-processes as a subset of Tπ threads. The communication rule $\mathsf{a}(x).p \mid \mathsf{a}\langle u \rangle \longrightarrow p\{x{:=}u\}$ in π can be simulated in Tπ by two downward movements, one anonymous communication in location a, and one upward movement. Address translations during these migrations have no accumulated effect, since we have $\mathsf{a}\oplus({}^{\mathsf{a}}\!\uparrow\!\oplus p) = p$. We believe that $lsd\pi$, or at least some trivial variant of it, is properly contained in Tπ also.

6 Conclusion

In this paper, we make some preliminary investigations in the formalization of an address-passing model in a distributed π-calculus with location hierarchy. Our approach of using addresses as the basic semantical entities seems to be quite natural in the presence of location hierarchy. The use of address translation makes the semantics quite simple and consistent with π and $lsd\pi$. With anonymous communication inside locations, we also unify channels and locations, which used to be two different classes in other distributed π-calculi.

The idea in this paper benefits a lot from existing works in distributed π-calculi and ambient calculi. Anonymous communications inside locations are

[5] The formal study of process behavior will be reported in another paper.

borrowed from ambients. However, in ambient calculi, ambients are mobile instead of fixed, and the idea of structured names doesn't make much sense. As early as in [11] (Chapter 14), Priami studied distributed name manager and relative addressing for π-calculus in a distributed setting using parallel composition. A simple form of address translation is first introduced for the two layer distributed calculus $lsd\pi$, carried out by dedicated substitutions of local and remote channels. Our address translation, however, is more general and works consistently within a tree of locations.

Location trees have also been explored in other process calculi close to π. In the local area π-calculus [4], a hierarchy of local areas is modeled and a same channel can have within its scope several disjoint local areas. The hierarchy, however, is predefined, and explicit process mobility is not modeled in the local area π-calculus. Locations in the distributed Join calculus (DJoin) [5] are also organized as a tree. However, DJoin use transparent remote communication and migration based on simple names. Locations and process mobility in DJoin are mainly used for modeling distribution and failure.

Another close work to Tπ is the π-calculus with polyadic synchronization (${}^e\pi$) [2]. Our calculus is even closer to ${}^e\pi$ if we use labels in communications and allow variables to appear as part of addresses. The notion of location and address translation, however, distinguish Tπ from ${}^e\pi$. These two calculi have different motivations and it is hard to express one in another, e.g. Tπ lacks the powerful name matching ability, while ${}^e\pi$ processes have to be coordinated in a centralized manner.

The syntax of Tπ studied in this paper is kept to its minimum to focus on the study of the static binding semantics of structured names. Extension could be made to express more features characterizing distributed mobile computation. Inspired by works like [7,4,14], we are investigating an extension of Tπ with *resource* names that are dynamically bound. Apart from anonymous communication, we now have nominal communication on these resource names. Resources are dynamically bound, and are not affected by address translation. We envisage embedding both Dπ and $lsd\pi$ in this version of Tπ.

Acknowledgments: Comments from Gérard Boudol, Ana Matos, and three anonymous referees improved this paper.

References

[1] Amadio, R. M., *On modelling mobility*, Theoretical Computer Science **240** (2000), pp. 147–176.

[2] Carbone, M. and S. Maffeis, *On the expressive power of polyadic synchronisation in pi-calculus*, in: *Proc. of the 9th International Workshop on Expressiveness in Concurrency (EXPRESS'02)*, ENTCS **68.2**, 2002.

[3] Cardelli, L. and A. D. Gordon, *Mobile ambients*, Theoretical Computer Science

240 (2000), pp. 177–213, an extended abstract appeared in *Proceedings of FoSSaCS '98*: 140–155.

[4] Chothia, T. and I. Stark, *A distributed calculus with local areas of communication*, in: *HLCL '00: Proceedings of the 4th International Workshop on High-Level Concurrent Languages*, Electronic Notes in Theoretical Computer Science **41.2** (2001).

[5] Fournet, C. and G. Gonthier, *The reflexive chemical abstract machine and the join-calculus*, in: *Proceedings of POPL '96*, ACM, 1996, pp. 372–385.

[6] Godskesen, J. C., T. Hildebrandt and V. Sassone, *A calculus of mobile resources*, in: *Proceedings of CONCUR'02*, LNCS **2421**, 2002, pp. 272–287.

[7] Hennessy, M. and J. Riely, *Resource access control in systems of mobile agents*, Journal of Information and Computation **173** (2002), pp. 82–120, an extended abstract appeared in *Proceedings of HLCL'98*, pages 3–17.

[8] Milner, R., *The polyadic π-calculus: A tutorial*, in: F. L. Bauer, W. Brauer and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, Series F **94**, NATO ASI (1993), available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[9] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, part I/II*, Journal of Information and Computation **100** (1992), pp. 1–77.

[10] Pierce, B. C. and D. Sangiorgi, *Typing and subtyping for mobile processes*, Mathematical Structures in Computer Science **6** (1996), pp. 409–454, an extract appeared in *Proceedings of LICS '93*: 376–385.

[11] Priami, C., "Enhanced Operational Semantics for Concurrency," Ph.D. thesis, Dipartimento di Informatica, Università di Pisa-Genova-Udine (1996), available as PhD number TD-08/96.

[12] Ravara, A., A. Matos, V. T. Vasconcelos and L. Lopes, *A lexically scoped distributed pi-calculus*, Di/fcul tr, DIFCUL (2002).

[13] Sangiorgi, D. and D. Walker, "The π-calculus: a Theory of Mobile Processes," Cambridge University Press, 2001.

[14] Schmitt, A., *Safe dynamic binding in the join calculus*, in: *Proceedings of the International IFIP Conference TCS 2002*, 2002, pp. 563–575.

[15] Schmitt, A. and J.-B. Stefani, *The m-calculus: a higher-order distributed process calculus*, in: *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2003), pp. 50–61, extended version as Rapport de Recherche RR-4361, INRIA Rhone-Alpes, January 2002.

[16] Vitek, J. and G. Castagna, *Seal: A framework for secure mobile computations*, in: H. Bal, B. Belkhouche and L. Cardelli, editors, *Internet Programming Languages*, LNCS **1686**, 1999, pp. 47–77.

[17] Wojciechowski, P. and P. Sewell, *Nomadic pict: Language and infrastructure design for mobile agents*, IEEE Concurrency **8** (2000), pp. 42–52.