



ELSEVIER

Information Processing Letters 80 (2001) 265–270

Information
Processing
Letters

www.elsevier.com/locate/ipl

Typing evolving ambients

Xudong Guan*, Yiling Yang, Jinyuan You

Department of Computer Science & Engineering, Shanghai Jiao Tong University, Shanghai 200030, People's Republic of China

Received 1 December 2000

Communicated by R. Backhouse

Abstract

This paper presents a type system that deals with the *type evolution problem* of the *ambient calculus*. It breaks the law that the `open` capability must introduce to the opening process the *exact* type information of the opened ambient. It can also serve as a general type architecture supporting type evolution. A typical usage of the type system is illustrated in the *immobile server* example. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Theory of computation; Type system; Ambient calculus

1. Introduction

The calculus of *Mobile Ambients* (MA) [3] is a model for *mobile computation* based upon the notion of *ambients*. An ambient may be thought of as a named location where computation happens. It is also the unit of movement, carrying all the processes and subambients within it: it can enter a neighboring ambient or go out of its parent ambient; it can be opened so that its contents merge into the opening processes. To obtain richer algebraic theories, variants of MA were proposed. *Mobile Safe Ambients* (SA) [9] introduced CCS style co-actions to control the actions of ambients. In SA, an ambient may traverse or open another ambient n only if at least one process inside n agrees. *RObust AMbients* (ROAM) [8] made further restrictions on reduction rules to guarantee the handshaking between actions and co-actions. Co-actions in ROAM can be consumed only by the

specified ambients. In this paper, we use *ambient calculus* to denote MA, SA, or ROAM, when the differences among them are not important.

Various type systems for the ambient calculus have been proposed. Topics studied so far include *message exchange type* and *mobility* [4,5,10], *security* [6,7,2], *threads* and *algebraic theory* [9,8], etc. In most of these type systems, the following law is uniformly adopted: the `open` capability must introduce to the opening process the *exact* type information of the opened ambient. For instance, in a type system that tracks the *mobility* of ambients, an ambient that opens a *mobile* ambient can not be typed as *immobile*. In fact, this law is too strict. In many cases, the unleashed process behaves differently from what it has done before. We call this the *type evolution problem* of the ambient calculus. This paper aims to devise an *evolving type system* (ETS) that supports type evolution. In addition, we try to make it a general ETS architecture, to which other type information can be easily added.

Our ETS is based on ROAM. It can also be adopted by SA with minor modifications (but not for MA, since

* Corresponding author.

E-mail addresses: guan-xd@cs.sjtu.edu.cn (X. Guan), yang-yl@cs.sjtu.edu.cn (Y. Yang), you-jy@cs.sjtu.edu.cn (J. You).

the $\overline{\text{open}}$ capability, which is essential in identifying type evolution, does not exist in MA). Moreover, as the type evolution problem is caused by the opening actions only, pure ROAM (that is, the calculus without message exchange) is enough to present our solution. The message exchange constructions of the full calculus only complicates the system.

The rest of this paper is organized as follows. Section 2 reviews the syntax and reduction semantics of pure ROAM. Section 3 presents a basic ETS called ETS-MT, which records only two attributes, *mobility* and *threads*. A typical *immobile server* example is given in Section 4 to demonstrate the application of ETS-MT. Finally goes the conclusion.

2. ROAM review

ROAM [8] is a variant of SA, with the parameters of co-actions participating in reduction just like those of the actions.

Let \mathcal{N} be a countable set of names ranged over by n, m . The set of processes \mathcal{P} (ranged over by P, Q) and the set of capabilities \mathcal{M} (ranged over by M)¹ are defined below using a BNF-like grammar:

$$P ::= \mathbf{0} \mid (vn : T)P \mid P \mid P \mid !P \mid M.P \mid n[P]$$

$$M ::= \varepsilon \mid \text{in } n \mid \text{out } n \mid \text{open } n \mid \overline{\text{in}} n \mid \overline{\text{out}} n \mid \overline{\text{open}} \mid M.M$$

Null process ($\mathbf{0}$), restriction with type value ($(vn : T)$), parallel composition (\mid), replication ($!$) and prefixing (\cdot) are standard. $n[P]$ stands for an ambient named n , with process P running inside. Restriction is the only name binder. $fn(P)$ (respectively $fn(M)$) denotes the set of free names in process P (respectively capability M). We often omit the trailing $\mathbf{0}$ in process $M \cdot \mathbf{0}$.

A capability M can be an empty string (ε), a string of capabilities ($M.M$), or a single action. There are six distinct types of actions in ROAM: entering/exiting ambient n ($\text{in } n/\text{out } n$), allowing ambient n to enter/exit ($\overline{\text{in}} n/\overline{\text{out}} n$), opening ambient n ($\text{open } n$), or allowing to be opened ($\overline{\text{open}}$). It is important that

¹ As long as syntax is concerned, removing ε and $M.M$ from \mathcal{M} will generate the same set \mathcal{P} . We keep them only to show that, with the ability to type string of actions, our type system can be extended easily to the full calculus with message exchange.

$\overline{\text{open}}$ does not specify who can consume it. Otherwise, a top level ambient can no longer be opened.

The reduction rules of ROAM are:

$$m[\text{in } n . P_1 \mid P_2 \mid n[\overline{\text{in}} m . Q_1 \mid Q_2] \rightarrow n[m[P_1 \mid P_2] \mid Q_1 \mid Q_2]$$

$$n[m[\text{out } n . P_1 \mid P_2] \mid \overline{\text{out}} m . Q_1 \mid Q_2] \rightarrow m[P_1 \mid P_2] \mid n[Q_1 \mid Q_2]$$

$$\text{open } n . P \mid n[\overline{\text{open}} . Q_1 \mid Q_2] \rightarrow P \mid Q_1 \mid Q_2$$

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \quad \frac{P \rightarrow P'}{(vn : T)P \rightarrow (vn : T)P'}$$

$$\frac{P \rightarrow P'}{n[P] \rightarrow n[P']} \quad \frac{P \equiv P' \quad P' \rightarrow P'' \quad P'' \equiv P'''}{P \rightarrow P'''}$$

In the last rule, the structural congruence relation (\equiv) is the least congruence relation that:

- is a commutative monoid for $\mathbf{0}$ and \mid ;
- allows to stretch the scope of restrictions, to permute restrictions, and to cancel a restriction followed only by $\mathbf{0}$:

$$n \notin fn(P) \Rightarrow (vn : T)(P \mid Q) \equiv P \mid (vn : T)Q$$

$$n \neq m \Rightarrow (vn : T)m[P] \equiv m[(vn : T)P]$$

$$(vn : T_n)(vm : T_m)P \equiv (vm : T_m)(vn : T_n)P$$

$$(vn : T)\mathbf{0} \equiv \mathbf{0};$$

- satisfies

$$!P \equiv !P \mid P, \quad !\mathbf{0} \equiv \mathbf{0}, \quad \varepsilon . P \equiv P, \quad \text{and}$$

$$(M.M') . P \equiv M . (M' . P).$$

3. Evolving type system

The main task of ETS is to record the different *type information* before and after type evolution. Many kinds of type information, such as message exchange type, mobility, threads, and security constraints, can be selectively recorded to yield different type systems. To obtain a general ETS architecture, we use an abstract symbol U called *pre-type* to denote the type information recorded. U can be customized to serve different purposes.

As a first attempt, we present ETS-MT, a basic ETS recording only *mobility* and *threads*, which are

essential to obtain certain properties of behaviour equivalence [9].

3.1. ETS-MT: grammar

We first list the grammar of ETS-MT below:

$Z ::= \underline{\vee}$	mobility:	<i>immobile</i>
$\quad \curvearrowright$		<i>mobile</i>
$Y ::= 0$	threads:	<i>none</i>
$\quad 1$		<i>single</i>
$\quad \omega$		<i>multiple</i>
$U ::= Z^Y$	pre-type	
$T ::= \perp$	process type	<i>invalid</i>
$\quad U$		<i>simple</i>
$\quad U[T]$		<i>evolving</i>
$W ::= \$$	capability type:	<i>hole</i>
$\quad U \cdot_t W$		<i>prefixing</i>
$\quad T _t W$		<i>composition</i>
$\quad U[W]$		<i>evolution</i>

In ETS-MT, the pre-type set \mathcal{U} is defined on two sets: the mobility set $\mathcal{Z} = \{\underline{\vee}, \curvearrowright\}$ (ranged over by Z) and the threads set $\mathcal{Y} = \{0, 1, \omega\}$ (ranged over by Y). A process is *mobile* if it may perform at least one mobility action (in or out), otherwise it is *immobile*. Threads, the number of concurrent capabilities a process may exhibit, can be *none* (e.g., 0), *single* (e.g., $\text{out } n . \text{in } m . \overline{\text{out}} p$), or *multiple* (e.g., $\text{in } n | \overline{\text{out}} m$ or $\text{in } m$).

Ambient names and processes have the same type notion called *process type*, denoted by T , $T \in \mathcal{T}$. \mathcal{T} is recursively defined on \mathcal{U} . Invalid processes are given type \perp . A process that never exhibits $\overline{\text{open}}$ has a *simple type*. Otherwise, it has an *evolving type* $U[T]$ indicating that the process has a simple type U before exercising $\overline{\text{open}}$ and has type T after that. U is called its *current type* and T its *future type*. For instance, process $\text{in } n . \overline{\text{open}} . \text{in } m$ is mobile and single-threaded (\curvearrowright^1) before $\overline{\text{open}}$. It becomes immobile and multi-threaded ($\underline{\vee}^\omega$) after that. As a result, it has type $\curvearrowright^1 [\underline{\vee}^\omega]$.

To facilitate the typing of processes formulated by prefixing (\cdot) and parallel composition ($|$), two type operators \cdot_t and $|_t$ are defined.

Firstly, consider the typing of process $M.P$ with $P : U[T]$ and $M : U'$ (we here assume M is simple,

that is, its effect can be described through some pre-type U'). Since M affects only the current type of $M.P$, $M.P$ is typed as $(U' \cdot_u U)[T]$, where \cdot_u is some function on pre-types, defined on top of \cdot_z and \cdot_y . The whole operation is denoted by \cdot_t .

$$\cdot_z : \mathcal{Z} \times \mathcal{Z} \rightarrow \mathcal{Z} \quad \cdot_y : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathcal{Y}$$

\cdot_z	$\underline{\vee}$	\curvearrowright	\cdot_y	0	1	ω
$\underline{\vee}$	$\underline{\vee}$	\curvearrowright	0	0	1	ω
\curvearrowright	\curvearrowright	\curvearrowright	1	1	1	ω
			ω	ω	ω	ω

$$\cdot_u : \mathcal{U} \times \mathcal{U} \rightarrow \mathcal{U}$$

$$Z_1^{Y_1} \cdot_u Z_2^{Y_2} = (Z_1 \cdot_z Z_2)^{(Y_1 \cdot_y Y_2)}$$

$$\cdot_t : \mathcal{U} \times \mathcal{T} \rightarrow \mathcal{T}$$

$$U \cdot_t \perp = \perp$$

$$U \cdot_t U' = U \cdot_u U'$$

$$U \cdot_t U'[T] = (U \cdot_u U')[T]$$

Secondly, consider the typing of process $P_1 | P_2$. There are four different cases to consider.

- (i) If both of them have simple types, say U_1 and U_2 , $P_1 | P_2$ is typed as $U_1 |_u U_2$, where $|_u$ is an operator like \cdot_u (defined on top of $|_z$ and $|_y$ instead).
- (ii) If one of them, say P_1 , has simple type U and the other has evolving type $U'[T]$, U is cast to both U' and T as P_1 may run either before or after the consumption of $\overline{\text{open}}$ in P_2 .
- (iii) If both of them have evolving types, $P_1 | P_2$ is difficult to type, since either of the two parallel $\overline{\text{open}}$ actions may be consumed first. We simply let this case to be invalid (\perp).²
- (iv) If one of them is invalid, the result is invalid too.

$$|_z : \mathcal{Z} \times \mathcal{Z} \rightarrow \mathcal{Z} \quad |_y : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathcal{Y}$$

$ _z$	$\underline{\vee}$	\curvearrowright	$ _y$	0	1	ω
$\underline{\vee}$	$\underline{\vee}$	\curvearrowright	0	0	1	ω
\curvearrowright	\curvearrowright	\curvearrowright	1	1	ω	ω
			ω	ω	ω	ω

²In fact, this case can also be typed, suppose $P_1 : T_1$, $P_2 : T_2$, let $T_3 = T_1 |_t T_2$, and let each evolving type be: $T_i = U_{i0}[U_{i1}[\dots[U_{in_i}]\dots]]$, $i = 1, 2, 3$, then $U_{3k} = \text{MAX}(U_{1i} |_u U_{2j})_{i+j=k}$, $k = 0..(n_1 + n_2)$. But this kind of non-determinism is rarely used.

$$\begin{aligned} |_u : \mathcal{U} \times \mathcal{U} &\rightarrow \mathcal{U} \\ Z_1^{Y_1} |_u Z_2^{Y_2} &= (Z_1 |_z Z_2)^{(Y_1 |_y Y_2)} \end{aligned}$$

$$\begin{aligned} |_t : \mathcal{T} \times \mathcal{T} &\rightarrow \mathcal{T} \\ U |_t U' &= U |_u U' \\ U |_t U'[T] &= (U |_u U')[U |_t T] \\ U'[T] |_t U &= (U' |_u U)[T |_t U] \\ U_1[T_1] |_t U_2[T_2] &= \perp \\ \perp |_t T &= \perp \\ T |_t \perp &= \perp \end{aligned}$$

Unlike other existing type systems for the ambient calculus, a *capability type* in ETS, W , is a context of process type. W is a type expression containing a hole ($\$$). When this hole is filled with some process type T , W will be evaluated to some other process type T' . This is nature, as each capability M needs a process P to form a new process $M.P$. $W(T)$ (respectively $W(W')$) denotes the result of replacing the hole in W with T (respectively W'). $W(T) \in \mathcal{T}$ (respectively $W(W') \in \mathcal{W}$).

ETS-MT also adopts a certain level of subtyping. The orders of \mathcal{Z} , \mathcal{Y} , \mathcal{U} , and \mathcal{T} are summarized below:

- They are all reflexive and transitive, and
- $\mathcal{Z}: \preceq \leq \prec, \mathcal{Y}: 0 \leq 1 \leq \omega$,

\mathcal{U} and \mathcal{T} :

$$\begin{aligned} Y_1^{Z_1} \leq Y_2^{Z_2} &\Leftrightarrow Y_1 \leq Y_2 \wedge Z_1 \leq Z_2 \\ U_1[T_1] \leq U_2[T_2] &\Leftrightarrow U_1 \leq U_2 \wedge T_1 \leq T_2 \end{aligned}$$

Process types with different depth of evolution (that is, with different nesting level of “[]”) are uncomparable.

3.2. ETS-MT: typing rules

There are three kinds of type judgements in ETS-MT: $\Gamma \vdash \diamond$ means that Γ is a good type environment; $\Gamma \vdash M : W$ means that M can be typed as W under Γ ; $\Gamma \vdash G : T$ means that n or P can be typed as T under Γ ($G \in \mathcal{G} = \mathcal{N} \cup \mathcal{P}$). The type environment Γ is a set whose elements have the form $n : T$. The *domain* of Γ is defined as $dom(\Gamma) = \{n \mid n : T \in \Gamma\}$.

The typing rules are divided into three parts.

Part 1. Typing rules for good environments and ambient names:

$$\frac{-}{\phi \vdash \diamond} \quad \frac{\Gamma \vdash \diamond, n \notin dom(\Gamma)}{\Gamma, n : T \vdash \diamond}$$

$$\frac{\Gamma, n : T \vdash \diamond}{\Gamma, n : T \vdash n : T}$$

Part 2. Typing rules for capabilities:

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \varepsilon : \$} \quad \frac{\Gamma \vdash n : T, M \in \{\text{inn}, \text{out } n\}}{\Gamma \vdash M : \overset{\circlearrowleft}{\cdot}_t \$}$$

$$\frac{\Gamma \vdash n : T, M \in \{\overline{\text{in}}n, \overline{\text{out}}n\}}{\Gamma \vdash M : \overset{\circlearrowright}{\cdot}_t \$}$$

$$\frac{\Gamma \vdash n : U[T]}{\Gamma \vdash \text{open } n : \overset{\circlearrowleft}{\cdot}_t (T |_t \$)} \quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \overline{\text{open}} : \overset{\circlearrowright}{\cdot}_t [\$]}$$

$$\frac{\Gamma \vdash M_1 : W_1, \Gamma \vdash M_2 : W_2}{\Gamma \vdash M_1.M_2 : W_1(W_2)}$$

Part 3. Typing rules for processes:

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \overset{\circlearrowleft}{\cdot}_t \perp} \quad \frac{\Gamma \vdash M : W, \Gamma \vdash P : T, W(T) \neq \perp}{\Gamma \vdash M.P : W(T)}$$

$$\frac{\Gamma \vdash P : T_1, \Gamma \vdash Q : T_2, T_1 |_t T_2 \neq \perp}{\Gamma \vdash P | Q : T_1 |_t T_2}$$

$$\frac{\Gamma \vdash P : T, T |_t T \neq \perp}{\Gamma \vdash !P : T |_t T} \quad \frac{\Gamma, n : T_1 \vdash P : T_2}{\Gamma \vdash (vn : T_1)P : T_2}$$

$$\frac{\Gamma \vdash n : T, \Gamma \vdash P : T}{\Gamma \vdash n[P] : \overset{\circlearrowleft}{\cdot}_t \perp} \quad \frac{\Gamma \vdash P : T_1, T_1 \leq T_2}{\Gamma \vdash P : T_2}$$

With the help of \cdot_t , $|_t$, and type context W , these rules are quite concise.

In Part 1, the first two rules judge a good environment. The other rule reads the type of a name in a good environment.

The rules in Part 2 cast the context-like capability types to the capabilities in \mathcal{M} . ε is typed as a hole. Simple movement capability (in , out , $\overline{\text{in}}$, or $\overline{\text{out}}$) is typed as $Z^1 \cdot_t \$$, where Z is either mobile (in , out) or immobile ($\overline{\text{in}}$, $\overline{\text{out}}$), open and $\overline{\text{open}}$ are treated specially, according to their different effects in type evolution. Consider process $\overline{\text{open}}.P$ with $P : T$. It has evolving type $\overset{\circlearrowright}{\cdot}_t [T]$, since it has a simple action ($\overline{\text{open}}$, which is immobile and single-threaded) before being opened and has type T after that. In process $\text{open } n.P$ with $P : T$, n must be an ambient having some evolving type $U[T']$. After the consumption of $\text{open } n$, there will be the parallel composition of P and the process P' come out of n . Since P' has the future type of n , T' , $\text{open } n.P$ has type $\overset{\circlearrowleft}{\cdot}_t (T' |_t T)$. The type of capability string is formulated through context substitution, which is still a context.

Typing rules for processes are given in Part 3. $\mathbf{0}$ has type $\underline{\vee}^0$, the bottom of simple type. Prefixing is typed through context substitution. $|_t$ is used to type parallel composition and replication. The typing of restriction is standard. Like $\mathbf{0}$, well-typed ambient (ambient name and the inside process have the same type) never issues any command to its outside world ($\underline{\vee}^0$). Finally, process types can be weakened according to the subtyping order.

From these rules, we can easily deduce that if $\Gamma \vdash P : T$, then $T \neq \perp$. Process with multiple $\overline{\text{open}}$ capabilities running in parallel is untypable, e.g., $!\overline{\text{open}}, \overline{\text{open}}. P \mid \overline{\text{open}}. Q$.

As an example, let $\Gamma = n : U_n[T_n], m : U_m$. We can derive $\Gamma \vdash \text{in } m. \overline{\text{in}} n. \text{open } n : \underline{\vee}^1 \cdot_t \underline{\vee}^1 \cdot_t \underline{\vee}^1 \cdot_t (T_n \mid_t \$)$. $\text{open } m$ is untypable under Γ , because m is a simple typed ambient (m can never be opened). $n[\mathbf{0}]$ is also untypable under Γ , for n is an evolving ambient (n must exhibit $\overline{\text{open}}$ sooner or later, which is impossible for $n[\mathbf{0}]$).

The soundness of ETS-MT is ensured by the following subject reduction theorem.

Theorem 1. *If $\Gamma \vdash P : T$ and $P \rightarrow Q$, then $\Gamma \vdash Q : T'$ and $T' \leq T$.*

The proof is by induction on the depth of the proof of $P \rightarrow Q$, a routine check similar to those of [8,5,10]. A detailed proof could be found in [12].

4. Immobile server

In this section, we employ a simple example to demonstrate the application of ETS-MT.

In this example, a network server is used to collect data brought by agents.

Server = $s[\overline{\text{in}} a. \text{open } a. \text{Collect}]$

Agent = $a[\text{in } s. \overline{\text{open}}. \text{Data}]$

We assume process **Collect** and process **Data** are both immobile.

As there are many agents arriving randomly, the server should be available at any time. This requires that the server be immobile. The server, however, must be typed as mobile without the introduction of type evolution, since it has the potential during its execution to open the mobile ambient a .

Server immobility can be easily achieved in ETS-MT. Suppose Γ is a good environment satisfying $\Gamma \vdash \text{Data} : \underline{\vee}^{Y_1}, \Gamma \vdash \text{Collect} : \underline{\vee}^{Y_2}, \Gamma \vdash s : T_s$, and $\Gamma \vdash a : T_a$. T_s and T_a are deduced as follows:³

- (1) $\Gamma \vdash s : T_s$ given
- (2) $\Gamma \vdash \text{in } s : \underline{\vee}^1 \cdot_t \$$ by (1)
- (3) $\Gamma \vdash \diamond$ given
- (4) $\Gamma \vdash \overline{\text{open}} : \underline{\vee}^1 [\$]$ by (3)
- (5) $\Gamma \vdash \text{in } s. \overline{\text{open}} : \underline{\vee}^1 \cdot_t \underline{\vee}^1 [\$]$ by (2) (4)
- (6) $\Gamma \vdash \text{Data} : \underline{\vee}^{Y_1}$ given
- (7) $\Gamma \vdash \text{in } s. \overline{\text{open}}. \text{Data} : \underline{\vee}^1 [\underline{\vee}^{Y_1}]$ by (5) (6)
- (8) $T_a = \underline{\vee}^1 [\underline{\vee}^{Y_1}]$ by (7)
- (9) $\Gamma \vdash \overline{\text{in}} a : \underline{\vee}^1 \cdot_t \$$ by (8)
- (10) $\Gamma \vdash \text{open } a : \underline{\vee}^1 \cdot_t (\underline{\vee}^{Y_1} \mid_t \$)$ by (8)
- (11) $\Gamma \vdash \text{Collect} : \underline{\vee}^{Y_2}$ given
- (12) $\Gamma \vdash \text{open } a. \text{Collect} : \underline{\vee}^1 \cdot_t (\underline{\vee}^{Y_1} \mid_t \underline{\vee}^{Y_2})$ by (10) (11)
- (13) $\Gamma \vdash \overline{\text{in}} a. \text{open } a. \text{Collect} : \underline{\vee}^1 \cdot_t (\underline{\vee}^{Y_1} \mid_t \underline{\vee}^{Y_2})$ by (9) (12)
- (14) $\Gamma \vdash \overline{\text{in}} a. \text{open } a. \text{Collect} : \underline{\vee}^\omega$ by (13)
- (15) $T_s = \underline{\vee}^\omega$ by (14)

By (15), we know that the server s can be typed as immobile.

In fact, applying both ETS-MT and the algebraic results in [9], the encoding of π -calculus into pure ambient calculus [11] can be proved pure algebraically.

5. Discussion

The type evolution problem has been approached in a number of different ways. The early type paper [5] introduced *objective moves* to let objective moving ambients be opened by immobile ambients. In [7], limited type evolution was acquired by requiring only one $\overline{\text{open}}$ capability to be present in the control process of an ambient. A recent paper [1] also introduces a form of type evolution to track *orderly communication*, during which the type of exchanged message can change

³ Since subtyping is allowed here, the results starting from (8) are the minimal value possible.

over time. Additionally, capabilities in their AC^+ calculus are assigned types that are also ‘behaviors with a hole within’, like the context-like capability type proposed here. Their work can be thought of as the type evolution of message exchange, while this paper focuses on the type evolution of mobility and threads. Moreover, the solution in this paper has a different motivation and tries to provide a general architecture for ETS. Following this ETS architecture, more type information can be stored in the pre-type to yield powerful type systems supporting type evolution.

Acknowledgements

We thank Yongqiang Sun and Pascal Zimmer for their precious help. Sincere thank goes to the three authors of [1] for pointing out the overlap between their work and ours. We are also grateful to the anonymous referee for the helpful comments.

References

- [1] T. Amtoft, A. Kfoury, S. Pericas-Geertsen, What are polymorphically-typed ambients?, in: European Symposium on Programming, April 2–6, 2001, to appear.
- [2] M. Bugliesi, G. Castagna, Secure safe ambients, in: Proceeding of POPL’01, 2001.
- [3] L. Cardelli, A. Gordon, Mobile ambients, in: Proc. FoS-SaCS’98, Lecture Notes in Comput. Sci., Vol. 1378, Springer, Berlin, 1998, pp. 140–155.
- [4] L. Cardelli, A. Gordon, Types for mobile ambients, in: Proc. POPL’99, ACM Press, New York, 1999, pp. 79–92.
- [5] L. Cardelli, G. Ghelli, A. Gordon, Mobility types for mobile ambients, in: Proc. ICALP’99, Lecture Notes in Comput. Sci., Vol. 1644, Springer, Berlin, 1999, pp. 230–239.
- [6] L. Cardelli, G. Ghelli, A. Gordon, Ambient groups and mobility types, in: Proc. TCS2000, Lecture Notes in Comput. Sci., Vol. 1872, Springer, Berlin, 2000, pp. 333–347.
- [7] M. Dezani-Ciancaglini, I. Salvo, Security types for mobile safe ambients, in: Proc. ASIAN’00, 2000.
- [8] X. Guan, Y. Yang, J. You, Making ambients more robust, in: Proc. Internat. Conf. on Software Theory and Practice, Beijing, China, 2000, pp. 377–384.
- [9] F. Levi, D. Sangiorgi, Controlling interference in ambients, in: Proc. POPL’00, Boston, MA, 2000, pp. 352–364.
- [10] P. Zimmer, Subtyping and typing algorithms for mobile ambients, in: Proc. FoSSaCS’2000, Lecture Notes in Comput. Sci., Vol. 1784, Springer, Berlin, 2000, pp. 375–390.
- [11] P. Zimmer, On the expressiveness of pure mobile ambients, in: Proc. EXPRESS’00, 2000.
- [12] Full version of this paper with proofs, available at: <http://go.163.com/~mobileambient/teafull.ps.zip>.