Formulating Mobile Languages

Tatsurou Sekiguchi and Akinori Yonezawa
Department of Information Science, Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113
Tel.: +81-3-3812-2111 ex. 4118 Fax: +81-3-5689-4365
e-mail: {cocoa,yonezawa}@is.s.u-tokyo.ac.jp

Abstract

This paper describes formal models of mobile computations, where dynamic transmission of code, data and execution environments is represented. Our models provide a foundation of mobile languages. We show that our models can represent the mobile mechanisms in Obliq, Telescript, Java, Facile and Kali-Scheme. The relation between code mobility and partial continuations are discussed. The correctness criteria for data mobility are presented and the correctness of the data movement is proved.

1 Introduction

Mobile languages[2, 17, 10, 11, 5, 4] encourages a programming style in which code, data and execution environments are dynamically moved. Behind this paradigm, there must be a presupposition that location-dependent operations and services should be exploited positively. This way of thinking is against the tendency in studies of distributed systems in a sense that it has been thought to be desirable that such a complicated matter is hidden from a user or a programmer and that local and remote access are transparent. The recent prosperity of mobile language systems, however, is proving that there would be interesting applications for such a programming style. A simple formal model for mobile language systems is strongly needed.

One of the difficulty in constructing a formal model for mobile language systems is the fact that a theoretical model is often missing the notion of locations so far because it is simpler. A typical instance of such a model is Linda's tuple space[3]. Several models[8, 12] based on modal logic where modality means locations are proposed, in which locations can be directly represented. But these models have a problem to describe existing mobile language systems such that it is hard to represent different copying policies for different mobile language systems. For instance, some system[16] duplicates all the values reachable from the target value, but Obliq[2] duplicates partly and the values that are not duplicated are referred to by remote references. In a modal logical system where modality means locations, copy operation must include operations on modality. Thus to compare different copying policies of mobile languages, different operations on modality are needed. This makes difficult to compare them in a common framework.

We take another approach. We introduce two simple models of distributed computations; one is based on untyped call-by-value λ-calculus with stores and the other is based on untyped imperative object calculus[1]. The basic idea behind them is *data movement protocols*[14] and *partial continuations*[7, 6]. Copying policies are specified by data movement protocols, which say how a value is moved to another site. We have investigated many mobile language systems and found out that 6 types are sufficient to represent copying policies in almost all systems. Data movement that may involve recursive transmission of values is formalized by *data movement relation*. We will show that the relation is correct in a sense that it preserves referring structures. The area of code and execution environments moved to another site is specified by partial continuations. We will show that an agent transmission operation can be considered as **escape**-and **abort**-operations[7].

$$
\begin{aligned}
p &\in \text{Place} \\
\tau &\in \text{DMP} ::= \textbf{copy} \mid \textbf{resident} \mid \textbf{carry} \mid \textbf{proper} \mid \textbf{takeaway} \mid \textbf{ubiq} \\
a &\in \text{Addr} \\
a@p &\in \text{Rem} = \text{Addr} \times \text{Place} \\
v &\in \text{Val} ::= () \mid p \mid a \mid a@p \mid \mathsf{go} \mid \mathsf{placeof} \mid \lambda x.e \\
e &\in \text{Exp} ::= v \mid x \mid ee \mid \mathsf{fix}\ x.e \mid \langle e \rangle \mid \mathsf{ref}_\tau e \mid !e \mid \mathsf{set}\ ee \\
s &\in \text{Store} = \text{Addr} \xrightarrow{\text{fin}} (\text{Val} \uplus \{\bot\}) \times \text{DMP} \\
C &::= [\,] \mid Ce \mid (\lambda x.e)C \mid \langle C \rangle \mid \mathsf{set}\ Ce \mid \mathsf{set}\ vC \mid !C \mid \mathsf{ref}_\tau C
\end{aligned}
$$

Figure 1: Domains

The rest of the paper is organized as follows: Section 2 introduces the basic setting of our framework and shows that several mobile languages are translated into our framework in a simple way. In Section 3, a model specialized for object-based systems is introduced. Section 4 discusses the relationship between code mobility and partial continuations while Section 5 show the correctness of data movement. Section 6 concludes this paper.

2 Imperative Mobile λ-Calculus

Since we want to catch the distinction between copy and sharing, in other words identity and location of values, to discuss the differences of mobility mechanisms varying widely depending on mobile language systems, an extension of call-by-value λ-calculus with stores is introduced. The expressions are lambda expressions extended by a fixed-point operator(fix), an agent expression($\langle\rangle$) and operations on stores. The operations on stores consist of initialization(ref_τ) where τ denotes a data movement protocol provided in this section, dereference(!) and assignment(set). The unit value () is the value of assignment(set). A place p abstracts a location where expressions are evaluated. $a@p$ denotes a remote reference referring to address a at place p. We use let-expressions as a syntactic macro such that $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \equiv (\lambda x.e_2)e_1$ and $e_1; e_2$ to denote $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ if $x \notin \text{FV}(e_2)$. Our previous paper[14] describes the semantics in detail. The semantic objects of the calculus is listed in Fig 1.

2.1 Preliminary definitions

$[y_1, .., y_n/x_1, .., x_n]$ denotes a substitution that replaces each x_i with y_i for $i = 1..n$. $\equiv$ is a syntactic equivalence. Let f be a function. $\text{Dom}(f)$ and $\text{Rng}(f)$ denote the domain and the range of f, respectively. $f[x \mapsto v]$ denotes a function such that

$$
(f[x \mapsto v])(y) = \begin{cases} v & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}
$$

The disjoint union of sets A, B is denoted by $A \uplus B$. We say a function g is an extension of f, noted $f \sqsubseteq g$, if there exists a function h such that $g = f \uplus h$. Both local and remote references are ranged over $\mathfrak{a}, \mathfrak{a}', ...$

$$
\begin{aligned}
s_p \vdash_p C[(\lambda x.e)v] &\longrightarrow C[[v/x]e], s_p && (\beta)\\
s_p \vdash_p C[\mathsf{fix}\ x.e] &\longrightarrow C[[(\mathsf{fix}\ x.e)/x]e], s_p && (\mathrm{fix})\\
s_p \vdash_p C[\langle v\rangle] &\longrightarrow C[v], s_p && (\mathrm{agent})\\
s_p \vdash_p C[\mathsf{ref}_\tau v] &\longrightarrow C[a], s_p[a \mapsto v^\tau] \quad \text{for some } a \notin \mathrm{Dom}(s_p). && (\mathrm{ref})\\
s_p \vdash_p C[!a] &\longrightarrow C[v], s_p \quad \text{if } s_p(a) = v^\tau. && (\mathrm{deref})\\
s_p \vdash_p C[\mathsf{set}\ a v] &\longrightarrow C[()], s_p[a \mapsto v^\tau] \quad \text{where } s_p(a) = v'^\tau \text{ for some } v' && (\mathrm{set})\\
s_p \vdash_p C[\mathsf{placeof}\ a] &\longrightarrow C[p], s_p && (\mathrm{place1})\\
s_p \vdash_p C[\mathsf{placeof}\ a@q] &\longrightarrow C[q], s_p && (\mathrm{place2})
\end{aligned}
$$

Figure 2: Operational semantics without go

2.2 Operational semantics

The sequential subset of the calculus is defined by call-by-value reduction semantics using contexts in Fig 2. Given a place p and a store s_p, the operational semantics associates an expression e with the reduced expression e' and the possibly updated store s'_p. This is denoted by $s_p \vdash_p e \longrightarrow e', s'_p$. Contexts are used to determine the redex of given expressions. The agent expression($\langle\rangle$) is just ignored in sequential reduction. placeof-operator takes out a place which a reference referring to.

Distributed and thus concurrent computation is represented as a transition semantics. A transition relation $\rightsquigarrow$ is between global states that are a pair of a finite map $S : \mathrm{Place} \stackrel{\mathrm{fin}}{\to} \mathrm{Store}$, and a multiset E of expressions with their places. The relation between local computations and distributed computations are established by the following rule:

$$\frac{S(p) \vdash_p e \longrightarrow e', s'}{(S, \{\vdash_p e\}) \rightsquigarrow (S[p \mapsto s'], \{\vdash_p e'\})}$$

This implies that any possible local computation can be a global transition. Let T be a finite map of stores and F be a multiset of execution states. The following rule implies that we can add unrelated stores and states to a transition freely.

$$\frac{(S, E) \rightsquigarrow (S', E') \quad \mathrm{Dom}(S) \cap \mathrm{Dom}(T) = \emptyset}{(S \uplus T, E \cup F) \rightsquigarrow (S' \uplus T, E' \cup F)}$$

2.3 Semantics of mobility

When a go-expression is evaluated, the innermost agent expression surrounding the go-expression moves to the destination and the rest of the expression without the agent still remains at the current place. Suppose we evaluate $C[\mathsf{go}\ q]$ at place p, and let $\langle A[\mathsf{go}\ q]\rangle$ be the innermost agent expression surrounding the go q such that $C[\mathsf{go}\ q] \equiv R[\langle A[\mathsf{go}\ q]\rangle]$ for some context R. Then $A[\mathsf{go}\ q]$ will move to the destination q and then be evaluated at q. We define that a go-expression is reduced to *the starting place* in view of agents. Thus, we have

$$C[\mathsf{go}\ q] \longrightarrow A[p].$$

On the other hand, the rest of the expression R stays at the current place. We define that a go-expression is reduced to *the destination place* in view of the expression surrounding the agent. Namely, we have

$$C[\mathsf{go}\ q] \longrightarrow R[q].$$

The operation of splitting an agent from the current context is precisely defined by the following function φ with an auxiliary function φ':

$$\begin{aligned}
\varphi(C) &= \varphi'([\,],[\,],C) \\
\varphi'(R,A,[\,]) &= (R,A) \\
\varphi'(R,A,Ce) &= \varphi'(R,A[[\,]e],C) \\
\varphi'(R,A,(\lambda x.e)C) &= \varphi'(R,A[(\lambda x.e)[\,]],C) \\
\varphi'(R,A,\langle C\rangle) &= \varphi'(R[\langle A\rangle],[\,],C)
\end{aligned}$$

provided that all the other expressions are ignored here for simplicity. We will discuss the relationship between φ and partial continuations in Section 4.

The rules for agent movement is formulated as follows:

$$\frac{\varphi(C) = (R,A) \quad \emptyset,\emptyset,s_q,A \Rightarrow_{s_p,q} \sigma,\theta,s'_q,A' \quad p \neq q}{([p \mapsto s_p, q \mapsto s_q], \{\vdash_p C[\mathsf{go}\ q]\}) \rightsquigarrow ([p \mapsto \sigma s_p, q \mapsto s'_q], \{\vdash_p \sigma R[q], \vdash_q A'[p]\})} \ \text{(go1)}$$

The innermost agent expression is splitted by φ and then is moved to the destination place by $\Rightarrow$. Finally, the effects of movement is applied to the current and the destination stores. In the case that the current and the destination places are common, the rule is provided as follows:

$$\frac{\varphi(C) = (R,A) \quad \emptyset,\emptyset,s_p,A \Rightarrow_{s_p,p} \sigma,\theta,s'_p,A'}{([p \mapsto s_p], \{\vdash_p C[\mathsf{go}\ p]\}) \rightsquigarrow ([p \mapsto \sigma s'_p], \{\vdash_p \sigma R[p], \vdash_p A'[p]\})} \ \text{(go2)}$$

2.4 Data movement protocols

Each address in a store is annotated by DMP(Data Movement Protocols), which describe how a stored value is handled when it is transmitted to another place. DMP is specified explicitly when a memory is allocated by ref-operation. The 6 types of DMP are *copy*, *resident*, *carry*, *proper*, *takeaway* and *ubiq*. A value of *copy* type indicates that the value is copied to the destination store. A *resident* type represents data which always stay at the current place, are never moved to other place, and are only referred to as a remote reference from the outside. Conversely, a *carry* type represents migration. A value of *carry* type is moved to the destination and is referred to by a *remote* reference from the previous place. A *proper* type represents a behavior of a value that never goes out and that cannot be referred to from the outside. Non-owner references in Telescript is of this type. A value of a *takeaway* type is considered to be inherently possessed by an agent. Finally, A *ubiq(uitous)* type is used to represent a common interface supposed to be facilitated by both the start and the destination place. Facile[11] has this type of values called proxy structures. These protocols are graphically represented in Fig 3.

Data movement operations using these protocols is formulated by effects of data movement. An effect is a quadruple (σ,θ,s,v) where

σ : the effect on the current store,

θ : the effect on the moving agent,

s : the possibly updated destination store, and

v : the moving data.

Data movement relation $\Rightarrow$ is the smallest relation on Effect $\times$ Store $\times$ Place $\times$ Effect satisfying the rules in Fig 4. In the form $\sigma,\theta,s,v \Rightarrow_{s_p,q} \sigma',\theta',s',v'$, the left-hand side expresses the effect before moving v, the right-hand side expresses the effect after the

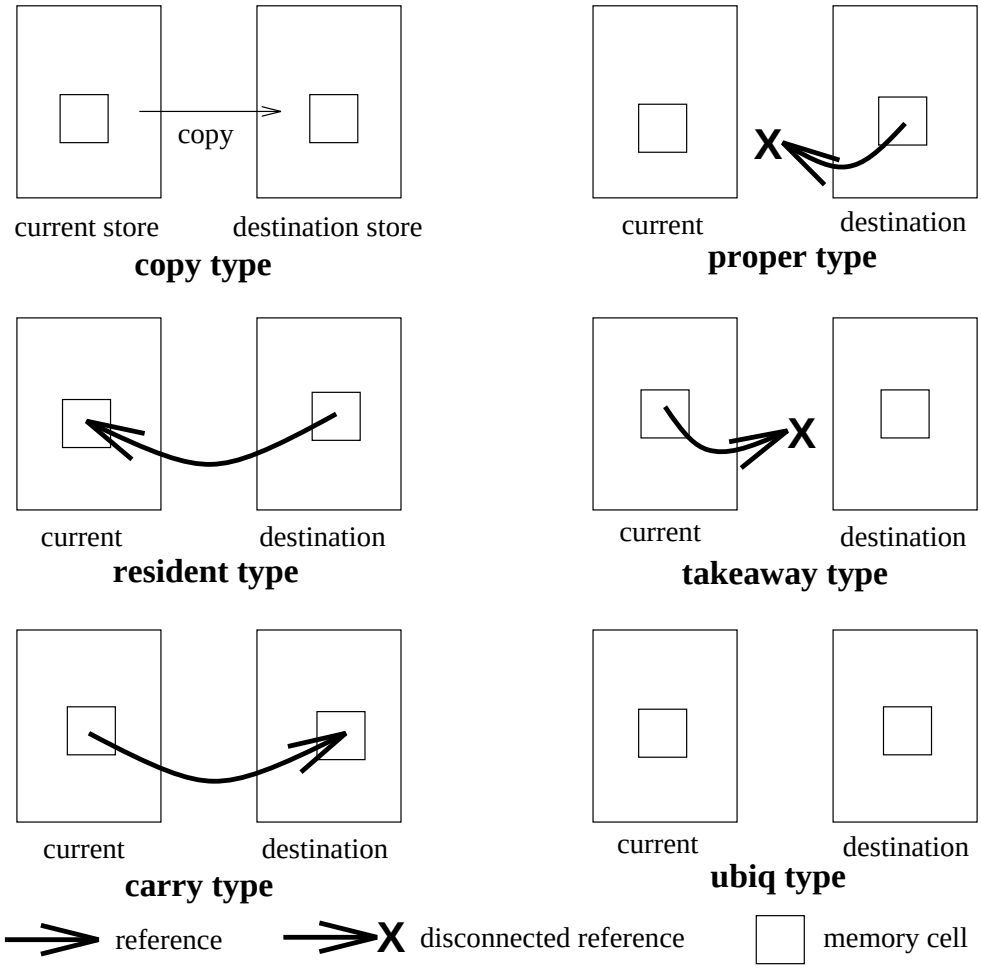

Figure 3: Data movement protocols

$$\frac{s_p(a) = v^{\mathbf{copy}} \quad \sigma, \theta[a \mapsto a'], s_q[a' \mapsto v'^{\mathbf{copy}}], v \Rightarrow_{s_p,q} \sigma', \theta', s'_q, v'}{\sigma, \theta, s_q, a \Rightarrow_{s_p,q} \sigma', \theta', s'_q, a'} \text{ (COPY)}$$

$$\frac{s_p(a) = v^{\mathbf{resident}}}{\sigma, \theta, s_q, a \Rightarrow_{s_p,q} \sigma, \theta[a \mapsto a@p], s_q, a@p} \text{ (RESIDENT)}$$

$$\frac{s_p(a) = v^{\mathbf{carry}} \quad \sigma[a \mapsto a'@q], \theta[a \mapsto a'], s_q[a' \mapsto v'^{\mathbf{carry}}], v \Rightarrow_{s_p,q} \sigma', \theta', s'_q, v'}{\sigma, \theta, s_q, a \Rightarrow_{s_p,q} \sigma', \theta', s'_q, a'} \text{ (CARRY)}$$

$$\frac{s_p(a) = v^{\mathbf{proper}}}{\sigma, \theta, s_q, a \Rightarrow_{s_p,q} \sigma, \theta[a \mapsto a'], s_q[a' \mapsto \bot], a'} \text{ (PROPER)}$$

$$\frac{s_p(a) = v^{\mathbf{takeaway}} \quad \sigma[a \mapsto \bot], \theta[a \mapsto a'], s_q[a' \mapsto v'^{\mathbf{takeaway}}], v \Rightarrow_{s_p,q} \sigma', \theta', s'_q, v'}{\sigma, \theta, s_q, a \Rightarrow_{s_p,q} \sigma', \theta', s'_q, a'} \text{ (TAKEAWAY)}$$

$$\frac{s_p(a) = v^{\mathbf{ubiq}}}{\sigma, \theta, s_q, a \Rightarrow_{s_p,q} \sigma, \theta[a \mapsto a], s_q, a} \text{ (UBIQ)}$$

provided that $a \in \text{Addr}$, $a \notin \text{Dom}(\theta)$ and $a' \notin \text{Dom}(s_q)$ for all the above rules.

$$\frac{r = q \quad a' \notin \text{Dom}(\theta)}{\sigma, \theta, s_q, a@r \Rightarrow_{s_p,q} \sigma, \theta[a' \mapsto a], s_q, a} \text{ (REM1)} \qquad \frac{r \neq q}{\sigma, \theta, s_q, a@r \Rightarrow_{s_p,q} \sigma, \theta, s_q, a@r} \text{ (REM2)}$$

$$\frac{a \in \text{Addr} \cap \text{Dom}(\theta)}{\sigma, \theta, s_q, a \Rightarrow_{s_p,q} \sigma, \theta, s_q, \theta a} \text{ (ADDR)}$$

$$\frac{\begin{array}{c}\{\mathfrak{a}_1, ..., \mathfrak{a}_n\} = \text{ADDR}(v) \\ \sigma_0, \theta_0, s_{q0}, \mathfrak{a}_1 \Rightarrow_{s_p,q} \sigma_1, \theta_1, s_{q1}, \mathfrak{a}'_1 \\ \vdots \\ \sigma_{n-1}, \theta_{n-1}, s_{qn-1}, \mathfrak{a}_n \Rightarrow_{s_p,q} \sigma_n, \theta_n, s_{qn}, \mathfrak{a}'_n\end{array}}{\sigma_0, \theta_0, s_{q0}, v \Rightarrow_{s_p,q} \sigma_n, \theta_n, s_{qn}, \theta_n v} \text{ (VAL)}$$

Figure 4: The movement relation $\Rightarrow$

movement, s_p is a store at a start place and q is a destination place. A superscript of a value in a store denotes DMP. We denote by $\mathrm{REF} : \mathrm{Context} \to \mathrm{Addr} \cup \mathrm{Rem}$ the set of local and remote references appeared in a context defined as follows:

$$\mathrm{REF}([]) = \mathrm{REF}(()) = \mathrm{REF}(p) = \mathrm{REF}(x) = \mathrm{REF}(\mathsf{go}) = \mathrm{REF}(\mathsf{placeof}) = \emptyset$$

$$\mathrm{REF}(a) = \{a\}, \qquad \mathrm{REF}(a@p) = \{a@p\}$$

$$\mathrm{REF}(\lambda x.e) = \mathrm{REF}(\mathsf{fix}\ x.e) = \mathrm{REF}(\langle e \rangle) = \mathrm{REF}(\mathsf{ref}_\tau e) = \mathrm{REF}(!e) = \mathrm{REF}(e)$$

$$\mathrm{REF}(e_1 e_2) = \mathrm{REF}(\mathsf{set}\ e_1 e_2) = \mathrm{REF}(e_1) \cup \mathrm{REF}(e_2).$$

Besides, the set of local addresses is denoted as $\mathrm{ADDR}(C) = \mathrm{REF}(C) \cap \mathrm{Addr}$ and the set of remote references as $\mathrm{REM}(C) = \mathrm{REF}(C) \cap \mathrm{Rem}$. Suppose that we move value v from place p to place q, if $\emptyset, \emptyset, s_q, v \Rightarrow_{s_p,q} \sigma, \theta, s'_q, v'$ holds, then after moving v the current and the destination stores will become σs_p and s'_q, respectively. The value v will change into θv in place q. There is an efficient algorithm that computes the right-hand side if the left-hand side, s_p and q are given.

2.5 Synchronous channel

We would like to reconsider a familiar concurrent construct in view of mobile computations. A simple synchronous channel is encoded by the following macros:

$$\begin{aligned}
\texttt{make_channel}() &\equiv \mathsf{ref}_{\mathbf{resident}}\ \texttt{empty} \\
\texttt{send}\ c\ v &\equiv \langle \mathsf{go}\ (\mathsf{placeof}\ c); \mathsf{set}\ c\ v \rangle \\
\texttt{receive}\ c &\equiv \mathsf{if}\ (!c) = \texttt{empty}\ \mathsf{then}\ (\texttt{receive}\ c)\ \mathsf{else}\ (!c)
\end{aligned}$$

where `empty` is a special constant representing the empty channel. `make_channel` macro creates a new address of resident type initialized the empty channel. *Send* macro stores a value at the address of given channel in the place where the channel was created. `receive` macro waits until a value will be stored in the channel. A receiver must be in the place where the channel was created.

2.6 Kali Scheme

Kali-Scheme[4] is a distributed implementation of Scheme interpreter that allows transmission of higher-order values including continuations. When values are transmitted to a remote place, they are in principle duplicated there provided that sharing within transmitted values is preserved. An operation for remove evaluation is provided. The expression (`remote_run!` place proc . args) evaluates a procedure and arguments at the current place, sends them to given place, and applies the procedure to the arguments. The operation is asynchronous and returns immediately with no return value. A proxy is available to distinguish local and remote data, which is an ordinary object of Scheme except that it remembers the place where it was created. If a proxy is transmitted to another place, the value is just duplicated there but one can update the original value by specifying its evaluation place explicitly like: (`remote_run!` (`proxy_creator` proxy) `set_proxy_value!` proxy new-value). We provide a translation $\mathcal{T}_K[\![]\!]$ from the essential subset of Kali-Scheme to our model as follows:

$$\begin{aligned}
\mathcal{T}_K[\![x]\!] &= x \\
\mathcal{T}_K[\![(\texttt{lambda}\ x.e)]\!] &= \lambda x.\mathcal{T}_K[\![e]\!] \\
\mathcal{T}_K[\![(e_1 e_2)]\!] &= \mathcal{T}_K[\![e_1]\!]\mathcal{T}_K[\![e_2]\!]
\end{aligned}$$

$$
\begin{aligned}
\mathcal{T}_K[\![\texttt{make_proxy}]\!] &= \lambda x.(\mathsf{ref}_{\mathbf{resident}}\ 0, \mathsf{ref}_{\mathbf{copy}}\ x) \\
\mathcal{T}_K[\![\texttt{proxy_creator}]\!] &= \lambda(\mathrm{gid}, \mathrm{val}).\mathrm{placeof\ gid} \\
\mathcal{T}_K[\![\texttt{proxy_value}]\!] &= \lambda(\mathrm{gid}, \mathrm{val}).!\mathrm{val} \\
\mathcal{T}_K[\![\texttt{set_proxy_value!}]\!] &= \lambda(\mathrm{gid}, \mathrm{val})v.\mathrm{set\ val}\ v \\
\mathcal{T}_K[\![\texttt{remote_run!}]\!] &= \lambda pfv.\langle \mathsf{go}\ p; f\ v\rangle
\end{aligned}
$$

The function `remote_run!` is quite similar to REV request[15] though REV is synchronous. Synchronous version of remote evaluation is constructed with synchronous channel as follows:

$$
\mathrm{reval}(e,p) \equiv \mathsf{let}\ c = \texttt{make_channel}(), _ = \langle \mathsf{go}\ p; \mathsf{send}\ c\ e\rangle\ \mathsf{in}\ \texttt{receive}\ c
$$

This macro defines evaluation of expression e at place p. To evaluate e at place p, a channel is created and the execution at the current place waits until a return value will be stored in the channel. An agent goes to place p, evaluates e there, and sends back the value to his birthplace through the channel.

2.7 Facile

Facile[11] is an extension of Standard ML added mobility, where there are two kinds of functions; transmissible functions and non transmissible ones. The compiler generates transmissible representations for transmissible functions. Non transmissible functions are never transmitted outside. The distinction of functions must be annotated by a programmer. A transmissible function is defined by the keyword `xfun` instead of `fun` for ordinary functions. A proxy structure accomplishes remote resource access in Facile. When an agent carrying signatures for the proxy structures is transmitted to a remote place, the destination place must provided their implementation and the signatures are linked to their implementation dynamically so that the agent can access local resources in that place. In facile, such kind of values are called ubiquitous values. The proxy structure of Facile is understood by means of ubiq type in our model. We provide a translation $\mathcal{T}_F[\![]\!]$ from a subset of Facile into our model as follows:

$$
\begin{aligned}
\mathcal{T}_F[\![x]\!] &= x \\
\mathcal{T}_F[\![\texttt{fn}\ x \Rightarrow e]\!] &= \mathsf{ref}_{\mathbf{proper}}\ (\lambda x.\mathcal{T}_F[\![e]\!]) \\
\mathcal{T}_F[\![(e_1 e_2)]\!] &= (!\mathcal{T}_F[\![e_1]\!])\mathcal{T}_F[\![e_2]\!] \\
\mathcal{T}_F[\![\texttt{fun}\ f x = e_1\ \texttt{in}\ e_2]\!] &= \mathsf{let}\ f = \mathsf{ref}_{\mathbf{proper}}\ \lambda x.\mathcal{T}_F[\![e_1]\!]\ \mathsf{in}\ \mathcal{T}_F[\![e_2]\!] \\
\mathcal{T}_F[\![\texttt{xfun}\ f x = e_1\ \texttt{in}\ e_2]\!] &= \mathsf{let}\ f = \mathsf{ref}_{\mathbf{copy}}\ \lambda x.\mathcal{T}_F[\![e_1]\!]\ \mathsf{in}\ \mathcal{T}_F[\![e_2]\!] \\
\mathcal{T}_F[\![\texttt{xstructure}\ s\ \texttt{in}\ e]\!] &= \mathsf{let}\ s = \mathrm{link}(s)\ \mathsf{in}\ \mathcal{T}_F[\![e]\!]
\end{aligned}
$$

where "link" is a mapping from variables to ubiq addresses that maps a name to its service. Suppose that $a_{\mathrm{printf}} \in \mathrm{Addr}$ is an entry point at which `printf` is implemented for each places. Then we have that $\mathrm{link}(\texttt{printf}) = a_{\mathrm{printf}}$. Though a proxy structure is an extension of ML's module system, it is treated as a function in this translation for simplicity.

2.8 Java class loader

Java has quite poor mobility mechanism by itself. A code of a class is not transmitted from a remote server until a method of an unloaded class is called. We describe the class loading mechanism of Java. For simplicity, we show the case of a function that is not

$$
\begin{aligned}
e ::= \;& p \mid x \mid \{l_i = \varsigma(x_i)e_i^{i\in 1..n}\}^{\tau} \mid e.l \mid e.l := \varsigma(x)e \mid e.\mathsf{place} \mid e.\mathsf{go}(e) \\
& \mid (\tau)e \mid \mathsf{clone}(e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
C ::= \;& [\,] \mid C.l \mid C.l := \varsigma(x)e \mid C.\mathsf{place} \mid C.\mathsf{go}(e) \mid v.\mathsf{go}(C) \mid (\tau)C \mid \mathsf{clone}(C) \\
& \mid \mathsf{let}\ x = C\ \mathsf{in}\ e \mid \langle C \rangle
\end{aligned}
$$

Figure 5: Expressions

copied until it is called. To describe such a code movement mechanism, the following stores suffice:

$$
\begin{aligned}
\text{server} &: [a \mapsto f^{\mathbf{resident}}] \\
\text{client} &: [\text{stub} \mapsto (\lambda x.\mathsf{let}\ d = \mathsf{set\ stub\ reval}(!(a@\text{server}), \text{server})\ \mathsf{in}\ !\text{stub}\ x)^{\mathbf{copy}}]
\end{aligned}
$$

We have a function f at the server place and a stub of the function at the client place. When the stub is invoked with a value v like !stub v, it copies the function f to the client place and then invoke f with v.

3 Imperative Mobile Object Calculus

In this section, an object version of mobile calculus is introduced, which is an extension of untyped imperative object calculus **impς**[1]. The syntax is provided in Fig 5.

An object $\{l_i = \varsigma(x_i)e_i^{i\in 1..n}\}^{\tau}$ is a record of methods $l_i = \varsigma(x_i)e_i$ where l_i is a method name and e_i is a method body. It is also the unit of transmission where τ is DMP. Method invocation $e.l$ executes the method with a reference of the self object as the argument. $e.l := \varsigma(x)e$ denotes method update. A special method invocation $e.\mathsf{place}$ evaluates to the place where e stays. A move operation $e.\mathsf{go}(e)$ transmits the current method and evaluates it at the destination place. A DMP-coercion operation $(\tau)e$ changes the object's DMP with τ. A cloning operation $\mathsf{clone}(e)$ creates a new object with the same methods as e. The semantics is defined in Fig 6. The core of the semantics except mobile constructs is equivalent to the one of **impς**.

An object version of synchronous channels is encoded as follows:

$$
\begin{aligned}
\$c.e \equiv\ & \mathsf{let\ chan} = \{\mathit{val} = \varsigma(x).\mathtt{empty}\}^{\mathbf{resident}} \\
& \mathsf{in\ let}\ c = \{\,\mathit{val} = \varsigma(x)\{\}, \\
& \qquad \mathit{update} = \varsigma(x)\mathsf{let}\ y = x.\mathit{val}\ \mathsf{in}\ c.\mathit{val} := \varsigma(x)y, \\
& \qquad \mathit{send} = \varsigma(x)x.\mathsf{go}(c.\mathsf{place}).\mathit{update}, \\
& \qquad \mathit{receive} = \varsigma(x)\mathsf{if}\ c.\mathit{val} = \mathtt{empty}\ \mathsf{then}\ x.\mathit{receive}\ \mathsf{else}\ c.\mathit{val}\}^{\mathbf{copy}} \\
& \mathsf{in}\ e
\end{aligned}
$$

A sender sets a value before sending by updating the *val* method of a channel $c.\mathit{val} := \varsigma(x)v$. Then it sends a value by *c.send*. A receiver waits until a value is set to the channel by *c.receive*.

3.1 Obliq

Obliq[2] is a simple distributed object-oriented language. The sequential subset of the semantics agrees with **imp$\lambda\varsigma$f**[1], which is an extension of **impς** with fields and procedures. It has a feature that one can specify the locations of values and controls in

$$\frac{a, a_i \notin \mathrm{Dom}(s_p) \quad \forall i \in 1..n}{s_p \vdash_p C[\{l_i = \varsigma(x_i) e_i^{i \in 1..n}\}^\tau] \longrightarrow C[a], s_p[a \mapsto \{l_i = a_i\}^\tau, a_i \mapsto \varsigma(x_i) e_i{}^{\mathbf{copy}}]} \text{ (obj)}$$

$$\frac{s_p(a) = \{l_i = a_i^{i \in 1..n}\}^\tau \quad s_p(a_j) = \varsigma(x_j) e_j{}^{\mathbf{copy}}}{s_p \vdash_p C[a.l_j] \longrightarrow C[\langle [a/x_j] e_j \rangle], s_p} \text{ (sel)}$$

$$\frac{s_p(a) = \{l_i = a_i^{i \in 1..n}\}^\tau}{s_p \vdash_p C[a.l_j := \varsigma(x) e] \longrightarrow C[a], s_p[a_j \mapsto \varsigma(x) e^{\mathbf{copy}}]} \text{ (update)}$$

$$s_p \vdash_p C[a.\mathsf{place}] \longrightarrow C[p], s_p \quad \text{(place1)}$$

$$s_p \vdash_p C[(a@q).\mathsf{place}] \longrightarrow C[q], s_p \quad \text{(place2)}$$

$$s_p \vdash_p C[\mathsf{let}\ x = v\ \mathsf{in}\ e] \longrightarrow C[[v/x]e], s_p \quad \text{(let)}$$

$$s_p \vdash_p C[\langle v \rangle] \longrightarrow C[v], s_p \quad \text{(agent)}$$

$$\frac{s_p(a) = v^{\tau'}}{s_p \vdash_p C[(\tau) a] \longrightarrow C[a], s_p[a \mapsto v^\tau]} \text{ (coerce)}$$

$$\frac{s_p(a) = \{l_i = a_i^{i \in 1..n}\}^\tau}{s_p \vdash_p C[\mathsf{clone}(a)] \longrightarrow C[a'], s_p[a' \mapsto \{l_i = a_i'^{i \in 1..n}\}^\tau, a_i' \mapsto s_p(a_i)]} \text{ (clone)}$$

$$\frac{\varphi(C) = (R, A) \quad \emptyset, \emptyset, s_q, a \Rightarrow \sigma, \theta, s_q', a' \quad p \neq q}{([p \mapsto s_p, q \mapsto s_q], \{\vdash_p C[a.\mathsf{go}(q)]\}) \rightsquigarrow ([p \mapsto \sigma s_p, q \mapsto s_q'], \{\vdash_q (\theta A)[a'], \vdash_p (\sigma R)[q]\})} \text{ (go1)}$$

$$\frac{\varphi(C) = (R, A) \quad \emptyset, \emptyset, s_p, a \Rightarrow \sigma, \theta, s_p', a'}{([p \mapsto s_p], \{\vdash_p C[a.\mathsf{go}(p)]\}) \rightsquigarrow ([p \mapsto \sigma s_p'], \{\vdash_p (\theta A)[a'], \vdash_p (\sigma R)[p]\})} \text{ (go2)}$$

Figure 6: Operational semantics

a completely implicit manner. The difference of execution sites is due to the difference between functions and methods. In Obliq, a function is always executed locally since a function is copied across sites as a function is a value. On the other hand a method is executed remotely since an object, and of course its methods, is referred as a remote reference from the outside of its resident site. The semantics of this remote method invocation can be defined in our mobile object calculus. Invocation of a method l for a remote object $a@q$ is defined as follows:

$$(a@q).l \stackrel{\text{def}}{=} \mathrm{RPC}(a@q, \varsigma(x)(a@q).l, c.receive)$$

where RPC is a macro as follows:

$$\begin{aligned}\mathrm{RPC}(o, E, R) \equiv \$c.\{\, & rpc = \varsigma(x)\mathsf{let}\ r = x.\mathsf{go}(o.\mathsf{place}).eval\ \mathsf{in}\ (c.val := \varsigma(y)r).send,\\ & eval = E,\\ & run = \varsigma(x)x.rpc; R\}^{\mathbf{copy}}.run\end{aligned}$$

With this macro, we have the following transition:

$$(S, \{\vdash_p (a@q).l\}) \rightsquigarrow^* (S', \{\vdash_p c.receive, \vdash_q \mathsf{let}\ r = a.l\ \mathsf{in}\ (c.val := \varsigma(y)r).send\})$$

where $\rightsquigarrow^*$ denotes the reflexive transitive closure of $\rightsquigarrow$. Obviously, the expression at place p waits a value though channel c while the expression at place q evaluates method l of object a and then sends the result though channel c.

The semantics of remote method update and cloning of a remote object is also defined as follows respectively:

$$\begin{aligned}(a@q).l := \varsigma(y)e &\stackrel{\text{def}}{=} \mathrm{RPC}(a@q, \varsigma(x)(a@q).l := \varsigma(y)e, c.receive)\\ \mathsf{clone}(a@q) &\stackrel{\text{def}}{=} \mathrm{RPC}(a@q, \varsigma(x)(\mathbf{copy})\mathsf{clone}(a@q).l, (\mathbf{resident})c.receive)\end{aligned}$$

3.2 Telescript

Telescript has ownership mechanism[9] that is used to specify the area of objects moved to another place together and for garbage collection. When some object goes to other place, all values owned by the object will be also taken there and all references pointing to non-owned values are nullified. The object ownership relation is defined by owner reference and non-owner reference. An object that is referred to by owner reference is considered to be owned by the referrer. We represent such a difference of ownership by introducing owner variables, which are disjoint with ordinary variables and noted by overlined variable $\overline{x}$. The set of all free owner variables in e is denoted by $\mathrm{FOV}(e)$. We assume that an object bound by an owner variable is considered to be owned by the referrer. Telescript ownership mechanism is encoded by the following syntactic translation $\mathcal{T}_T[\![]\!]$ on expressions:

$$\begin{aligned}\mathcal{T}_T[\![p]\!] &= p\\ \mathcal{T}_T[\![x]\!] &= x\\ \mathcal{T}_T[\![\overline{x}]\!] &= \overline{x}\\ \mathcal{T}_T[\![\{l_i = \varsigma(x_i)e_i^{i\in 1..n}\}^\tau]\!] &= \{l_i = \mathcal{T}_T[\![\varsigma(x_i)e_i]\!]^{i\in 1..n}\}^\tau\\ \mathcal{T}_T[\![e.l]\!] &= \mathcal{T}_T[\![e]\!].l\\ \mathcal{T}_T[\![e.l := \varsigma(x)e']\!] &= \mathcal{T}_T[\![e]\!].l := \mathcal{T}_T[\![\varsigma(x)e']\!]\\ \mathcal{T}_T[\![e.\mathsf{place}]\!] &= \mathcal{T}_T[\![e]\!].\mathsf{place}\\ \mathcal{T}_T[\![e.\mathsf{go}(e')]\!] &= \mathcal{T}_T[\![e]\!].\mathsf{go}(\mathcal{T}_T[\![e']\!])\end{aligned}$$

$$
\begin{aligned}
\mathcal{T}_T[\![(\tau)e]\!] &= (\tau)\mathcal{T}_T[\![e]\!] \\
\mathcal{T}_T[\![\mathsf{clone}(e)]\!] &= \mathsf{clone}(\mathcal{T}_T[\![e]\!]) \\
\mathcal{T}_T[\![\mathsf{let}\ x = e\ \mathsf{in}\ e']\!] &= \mathsf{let}\ x = \mathcal{T}_T[\![e]\!]\ \mathsf{in}\ \mathcal{T}_T[\![e']\!] \\
\mathcal{T}_T[\![\varsigma(x)e]\!] &= \varsigma\psi_p(\mathrm{FV}(e) \setminus \mathrm{FOV}(e)); \psi_t(\mathrm{FOV}(e)); \mathcal{T}_T[\![e]\!]
\end{aligned}
$$

where ψ_p and ψ_t are auxiliary maps as follows:

$$
\begin{aligned}
\psi_p(\{x_1, ..., x_n\}) &= (\mathbf{proper})x_1; ...; (\mathbf{proper})x_n \\
\psi_t(\{x_1, ..., x_n\}) &= (\mathbf{takeaway})x_1; ...; (\mathbf{takeaway})x_n
\end{aligned}
$$

It is easy to see that every object bound by an owner variable has **takeaway** type.

4 Code mobility and partial continuations

The code movement mechanism may seem ad hoc, but it can be understood naturally in view of partial continuations[6, 7]. Though a concurrent system is required to express agent movement since we must have a start and a destination places, since we want to concentrate on control structures in this section, the reduction rules are modified so that it becomes a sequential system. Also, stores are omitted for the same reason. The grammar of expressions and contexts, and the reduction rules are as follows:

$$
\begin{aligned}
e &::= \ \mathsf{go} \mid \lambda x.e \mid ee \mid \langle e \rangle \\
C &::= \ [\,] \mid Ce \mid (\lambda x.e)C \mid \langle C \rangle
\end{aligned}
$$

$$
\begin{aligned}
C[(\lambda x.e)v] &\longrightarrow C[[v/x]e] && (\beta) \\
C[\langle v \rangle] &\longrightarrow C[v] && (\text{AGENT}) \\
C[\mathsf{go}] &\longrightarrow R[\lambda x.A[x]] && (\text{GO})
\end{aligned}
$$

where $\varphi(C) = (R, A)$ and x is a fresh variable in the rule GO. A translation mapping $\mathcal{T}_P[\![]\!]$ defined below from the above calculus to Felleisen's framework[7] of **control** and **prompt** shows the relationship between code mobility and partial continuations.

$$
\begin{aligned}
\mathcal{T}_P[\![x]\!] &= x \\
\mathcal{T}_P[\![e_1 e_2]\!] &= \mathcal{T}_P[\![e_1]\!]\mathcal{T}_P[\![e_2]\!] \\
\mathcal{T}_P[\![\lambda x.e]\!] &= \lambda x.\mathcal{T}_P[\![e]\!] \\
\mathcal{T}_P[\![\mathsf{go}]\!] &= (\mathbf{escape}\ k\ k) \\
\mathcal{T}_P[\![\langle e \rangle]\!] &= (\#\mathcal{T}_P[\![e]\!])
\end{aligned}
$$

An agent-expression corresponds to a prompt expression. We can prove the following theorem:

Theorem 4.1. *If $e \longrightarrow e'$ then $\mathcal{T}_P[\![e]\!] \Rightarrow^* \mathcal{T}_P[\![e']\!]$. where $\Rightarrow^*$ denotes the reflexive transitive closure of reduction relation $\Rightarrow$ of Felleisen's system.*

Intuitively, go-expression can be interpreted as transmitting the partial continuation to the destination place in view of the agent, and as aborting operation up to the prompt expression in view of the current place. Queinnec proposes **splitter** operator[13] that composes abort and escape operations. The semantics of go is familiar with that kind of operators.

5 Correctness of the data movement relation

This section shows the correctness of the data movement relation $\Rightarrow$ in a sense that it preserves referring structures. We say that a store is well-formed if it is locally closed in terms of references and that a global store, which is a finite map from places to stores, is well-formed if it is closed in terms of every kind of references and ubiquitous values are implemented for each stores.

Definition 5.1 (Well-formed global store). Let S be a global store. S is well-formed if

1. each store is well-formed ($\forall s_p \in \mathrm{Rng}(S).\textsc{Addr}(\mathrm{Rng}(s_p)) \subseteq \mathrm{Dom}(s_p)$),
2. closed w.r.t. remote references ($\forall s_p \in \mathrm{Rng}(S).\textsc{Rem}(\mathrm{Rng}(s_p)) \subseteq \mathrm{ADom}(S)$) where $\mathrm{ADom}(S) = \{a@p | p \in \mathrm{Dom}(S), a \in \mathrm{Dom}(S(p))\}$, and
3. ubiquitous values are really ubiquitous ($\forall s_p \in \mathrm{Rng}(S).\forall a \mapsto v^{\mathbf{ubiq}} \in s_p.\forall s_q \in \mathrm{Rng}(S).a \in \mathrm{Dom}(s_q)$).

We show that transmitted values are correctly duplicated by the fact that it preserves referring structure, which is a sequence of local addresses ending up with a value or $\perp$. Every neighboring pair of address in a referring structure denotes that the former refers to the latter.

Definition 5.2 (Referring structure). Let S be a global store. Referring structure $\mathcal{R}_S$ is defined as follows:

$$\mathcal{R}_S(a,p) = \begin{cases} a\perp & \text{if } S(p)(a) = \perp \\ a\perp & \text{if } S(p)(a) = v^{\mathbf{ubiq}} \\ a\mathcal{R}_S(a',p) & \text{if } S(p)(a) = a'^{\tau} \\ a\mathcal{R}_S(a',p') & \text{if } S(p)(a) = a'@p'^{\tau} \end{cases}$$

$$\mathcal{R}_S(a@p,p') = \mathcal{R}_S(a,p)$$

$$\mathcal{R}_S(v,p) = v \qquad \text{if } v \notin \mathrm{Addr} \cup \mathrm{Rem}$$

The length of a referring sequence may be infinite but a sequence can be represented by a regular expression actually because a store has only finite elements and S is finite.

We consider two referring structures r, r' are equivalent, noted $r \approxeq_\theta r'$, if their components are identical or there exists a one-to-one correspondence θ between them. But referring structure may be disconnected suddenly when a value is transmitted if there is an address of **proper** type or **takeaway** type. In that case, the rest of the referring structure is not concerned to the equivalence.

Definition 5.3 (Equivalence of referring structures). $\approxeq$ is the smallest relation such that:

$$r \approxeq_\theta \perp \text{ and } \perp \approxeq_\theta r \text{ for any } r$$

$$a@p \approxeq_\theta a@p$$

$$v \approxeq_\theta v' \qquad \text{if } \theta v = v' \text{ or } v = v'$$

$$ar \approxeq_\theta a'r' \qquad \text{if } (\theta a = a' \text{ or } a = a') \text{ and } r \approxeq_\theta r'$$

We claim the correctness of $\Rightarrow$ in the sense of the following theorem. Note that referring structures are compared on the start store before movement and the destination store after movement.

Theorem 5.1 (Preservation of referring structure). *Let S be a well-formed global store $[p \mapsto s_p, q \mapsto s_q] \cup S_0$. Let A be a context such that $REF(A) \subseteq \{\bot\} \cup Dom(s_p) \cup ADom(S)$. If $\emptyset, \emptyset, s_q, A \Rightarrow_{s_p,q} \sigma, \theta, s'_q, A'$ then we have that*

$$\forall \mathfrak{a} \in ADDR(A).\mathfrak{a}' \in A' \wedge \mathcal{R}_{S'}(\mathfrak{a}, p) \approx_\theta \mathcal{R}_{S'}(\mathfrak{a}', q)$$

where $\mathfrak{a}' = \theta\mathfrak{a}$ and $S' = [p \mapsto s_p, q \mapsto s'_q] \cup S_0$.

The proof is by induction on the derivation of $\Rightarrow$. The key is preservation of well-formedness.

By changing the definition of referring structure $\mathcal{R}$, we can prove another property of $\Rightarrow$. We show that values on *local* stores are equivalent by the following referring structure $\mathcal{R}'$:

$$\mathcal{R}'_S(a,p) = \begin{cases} a\bot & \text{if } s_p(a) = \bot \\ a\bot & \text{if } s_p(a) = v^{\mathbf{ubiq}} \\ a\mathcal{R}'_S(a',p) & \text{if } s_p(a) = a'^{\tau} \\ a\bot & \text{if } s_p(a) = a'@p'^{\tau} \end{cases}$$

$$\mathcal{R}'_S(a@p,p') = \bot$$

$$\mathcal{R}'_S(v,p) = v \qquad \text{if } v \notin \mathrm{Addr} \cup \mathrm{Rem}$$

This referring structure does not advance a remote reference, thus it represents values on local stores.

Theorem 5.2 (Guarantee of carrying possessions). *Let S be a well-formed global store $[p \mapsto s_p, q \mapsto s_q] \cup S_0$. Let A be a context such that $REF(A) \subseteq \{\bot\} \cup Dom(s_p) \cup ADom(S)$. If $\emptyset, \emptyset, s_q, A \Rightarrow_{s_p,q} \sigma, \theta, s'_q, A'$ then we have that*

$$\forall \mathfrak{a} \in ADDR(A).\mathfrak{a}' \in A' \wedge \mathcal{R}'_{S'}(\mathfrak{a}, p) \approx_\theta \mathcal{R}'_{S'}(\mathfrak{a}', q)$$

where $\mathfrak{a}' = \theta\mathfrak{a}$ and $S' = [p \mapsto s_p, q \mapsto s'_q] \cup S_0$.

5.1 Copying garbage collection

Code movement is closely related to partial continuations. On the other hand, data movement subsumes copying garbage collection. Suppose that all references have copy types. Consider start store as old space and destination store as new space. Then data movement represents garbage collection procedure. Moreover, the proof of correctness shows the correctness of copying garbage collection. A weak pointer is expressed by proper type since a weak pointer is nullified once garbage collection occurs.

6 Conclusion

We have presented simple models of distributed and mobile computations. Our framework can encode various mechanisms in many mobile language systems as shown. Data movement is specified by data movement protocols, and code and execution states movement are specified by partial continuations. Data movement is represented by data movement relation $\Rightarrow$. We also showed its correctness.

We are now planning to incorporate security and authentication mechanisms into the framework, which are intentionally ignored in this study. A type system that captures some aspects of mobile computation would be interesting.

7 Acknowledgments

We would like to thank Luca Cardelli for his comments on our previous paper, who suggested the idea of mobile object calculus. We would also like to thank Christian Queinnec for pointing out a connection between code mobility and partial continuations.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Luca Cardelli. A Language with Distributed Scope. In *Conference Record of the 22th Symposium on Principles of Programming Languages*, pages 286–297, 1995.

[3] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[4] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-Order Distributed Objects. In *ACM Transactions on Programming Languages and Systems*, 1995. `http://www.neci.nj.nec.com/PLS/kali.html`.

[5] Colusa. Omniware: A Universal Substrate for Mobile Code. 1995. Colusa Software white paper. Available at `http://www.colusa.com`.

[6] Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.

[7] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract Continuations: A Mathematical Semantics for Handling Functional Jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, 1988.

[8] Cédric Fournet and Georges Gonthier. A Calculus of Mobile Agents. In *CONCUR'96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, 1996.

[9] General Magic Inc. *Telescript Programming Guide*. Version 1.0 alpha 2.

[10] James Gosling and Henry McGilton. The Java Language Environment. 1995. White paper. Sun Microsystems. Available at `http://java.sun.com/`.

[11] Frederick Colville Knabe. *Language Support for Mobile Agents*. PhD thesis, Computer Science Carnegie Mellon University, 1995.

[12] Naoki Kobayashi, Toshihiro Shimizu, and Akinori Yonezawa. Distributed Concurrent Linear Logic Programming. submitting.

[13] Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *Conference Record of POPL'91: 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–184, 1991.

[14] Tatsurou Sekiguchi and Akinori Yonezawa. A Calculus with Code Mobility. In *Proceedings of Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, 1997. Available at `ftp://yl.is.s.u-tokyo.ac.jp/pub/papers/cocoa-fmoods97-a4.ps.gz`.

[15] James W. Stamos and David K. Gifford. Remote Evaluation. In *ACM Transactions on Programming Languages and Systems*, volume 12, pages 537–565, 1990.

[16] Markus Straßer, Joachim Baumann, and Fritz Hohl. Mole – a java based mobile agent system. *ECOOP'96 Workshop on Mobile Object Systems*, 1996.

[17] James E. White. Mobile Agents. In Jeffrey Bradshaw, editor, *Software Agents*. The MIT Press, 1996.