A Study on Mobile Language Systems

By

Tatsurou Sekiguchi
cocoa@is.s.u-tokyo.ac.jp

April 1999

A Dissertation Submitted to
Department of Information Science
Graduate School of Science
The University of Tokyo

In Partial Fulfillment of the Requirements
For the Degree of Doctor of Science

Contents

1 Introduction **13**
- 1.1 What is Mobile Computation? . . . 13
- 1.2 A Remark on our Methodology to Investigate Mobile Computation . 15
- 1.3 Mobile Computation and the Economy of Programming . . . 17
- 1.4 An Overview and Contributions of This Thesis . . . 20
- 1.5 The Definitions of Mathematical Symbols and Notations . . . 20

2 A Calculus with Code Mobility **23**
- 2.1 Calculus Overview . . . 23
  - 2.1.1 Code Mobility . . . 23
  - 2.1.2 Data Mobility . . . 25
- 2.2 Operational Semantics . . . 27
  - 2.2.1 Local Semantics . . . 27
  - 2.2.2 Formulation of Data Transfer . . . 30
  - 2.2.3 Formulation of Distributed Computation . . . 33
  - 2.2.4 Formulation of Evaluation of Go Expressions . . . 34
- 2.3 Expressive Power . . . 34
  - 2.3.1 Remote Evaluation . . . 35
  - 2.3.2 Remote Procedure Call . . . 35
  - 2.3.3 Encoding Obliq RMI . . . 36
  - 2.3.4 Encoding Java Class Loader . . . 37
  - 2.3.5 Encoding Kali-Scheme Proxy Values . . . 38
  - 2.3.6 Encoding Telescript Ownership Mechanism . . . 38
- 2.4 Controllable Migration Transparency . . . 39
  - 2.4.1 *Partial* Continuation . . . 39
  - 2.4.2 Felleisen's Notion of Partial Continuation . . . 40
  - 2.4.3 Danvy's Notion of Partial Continuation . . . 41
  - 2.4.4 A Sequential Subset of our Mobile Calculus . . . 42
  - 2.4.5 Translation to Felleisen's System . . . 43
  - 2.4.6 Translation to Danvy's System . . . 45
  - 2.4.7 Application — Implementing our Mobile Calculus by Using the Extended CPS . . . 46
  - 2.4.8 Application — Optimizing the CPS Implementation . . . 47
- 2.5 Migration Transparency on Data . . . 47
  - 2.5.1 An Infinite Tree as a Representation of a Data Structure Embedded in the Heaps . . . 48
  - 2.5.2 Operations on Trees . . . 48
  - 2.5.3 From a Value to its Infinite Expansion . . . 49

2.5.4 Equivalence between Values in the Heaps 51
2.5.5 Soundness and Completeness 52
2.6 Related Work 56
2.6.1 Core Facile 56
2.6.2 A Model of Distributed Memory Management 56
2.6.3 Mobile Calculi as Extensions of Concurrent Calculi 57
2.7 Features not Supported in the Calculus 57
2.7.1 Module Mechanism and Variable-Binding 57
2.7.2 Hierarchical Locations 58
2.7.3 General Pointer Forwarding 58

3 A Programming Language for Mobile Computation 59
3.1 Initiating Migration by Using Go Primitive 59
3.2 Controlling Transparency by Using Undock Primitive 60
3.3 Declaring Method by Using Migratory Primitive 61
3.4 Creating Objects with Specifying Data Movement Protocols 61

4 An Implementation of the Programming Language 63
4.1 Background and Objectives 63
4.2 Overview of Our Transparent Migration 65
4.2.1 Saving the Execution State 65
4.2.2 Transmitting the Execution State 66
4.2.3 Restoring the Execution State 66
4.3 Resuming a Method 66
4.3.1 Preprocessings 66
4.3.2 Jump Facility 67
4.3.3 Unfolding Technique 68
4.4 Saving and Restoring Local Variable Values 69
4.4.1 The Arachne Scheme 69
4.4.2 Our Scheme 69
4.5 Reconstructing Stack of Method Invocations 70
4.6 Translating Other Program Constructs 71
4.6.1 Translating an Undock Statement 71
4.6.2 Translating a Synchronized Statement 71
4.6.3 Translating a finally clause 71
4.7 Migration Server 72
4.8 Limitations due to Source-Level Translation 72
4.9 Performance 73
4.10 Sample Mobile Applications 74
4.11 Related work and a need of extending JavaRMI for further development 76
4.12 The Translated Fibonacci Function 76

5 Conclusion 79

List of Figures

2.1 Domains. 28
2.2 The operational semantics for local evaluation. 30
2.3 The rules for address transfer. 32
2.4 The rules for data transfer. 33

3.1 Conceptual diagram of a stack. 61

4.1 Applications migrating between two computers. 75

List of Tables

4.1 Execution performance and growth in byte code size of programs transformed by our scheme. 73
4.2 Performance of Arachne scheme. 74

Abstract

The mobile form of distributed computation, called *mobile computation*, is investigated from the viewpoints of computational model, programming language and implementation. The emphasis of this thesis compared with the related work is on analyzing *existing* mobile language systems from a rather *implementational* point of view. The contribution of this thesis is threefold. (1) We present a computational model in which dynamic transference of code, data and execution state can be represented. Several mobility mechanisms in existing mobile language systems are analyzed in the model. (2) Based on the analysis, a programming language for mobile computation is designed. The distinctive feature of the language is the facility to control the *transparency* of migration. (3) An implementation of the mobile programming language is presented. The language is implemented as an extension of the Java language. The implementation is so portable that the program runs on any standard Java interpreter and can be compiled by any standard Java just-in-time (JIT) compiler. A program in the mobile language runs as fast as the non-mobile execution of the program.

Acknowledgment

I would like to express my deep gratitude to Akinori Yonezawa, who continuously supported and encouraged me during this work. I also profoundly thank the members of the Yonezawa group, who gave me a plenty of constructive comments and suggestions.

Chapter 1

Introduction

1.1 What is Mobile Computation?

We would like to dispense with describing the current status of mobile computation because it is exceedingly active, widespread not only among computer maniacs but also among naive users of computers (although they might be unaware of their commitment to mobile computation). Besides, it includes various topics from academic, research-level ones to commercial applications. Even when the domain of interest is restricted to academic issues, it comprises at least the fields of operating system, hardware infrastructure such as wireless and wired network [14, 31], and programming languages. More concretely, the whole issue comprises at least:

- the design of a novel IP packet format for flexible routing and error recovery [78],
- hardware memory protection mechanism for protecting a virtual machine from malicious external agents [59],
- design of bytecode for enabling efficient execution for heterogeneous environment [3, 23],
- dynamic compilation, dynamic code generation for efficient execution of mobile agents,
- theories for mobile agents to retain security properties [49, 73, 97], and
- application of mobility for quality of service (QoS) [77], workflow [15] and Internet chat [80].

In addition to the vastitude of relevant research fields, there is a terminological and conceptual confusion concerning its fundamental component. The mobility-related terms carry two different connotations depending on the research area. For those who are familiar with wireless and wired network, and communication protocols, mobility often implies physical mobility, the mobility in the real world. The application domain of physical mobility includes PDA (personal digital assistants) using GPS [66], support for laptop and notebook computers [38, 53, 54, 57], and reduction in energy consumption [50, 67]. On the other hand, for those who familiar with operating system and programming languages, mobility often implies software mobility. This is the reason why Luca Cardelli proposed that the former form of

mobility should be designated as *mobile computing*, while the latter form of mobility should be designated as *mobile computation*.

There is another terminological confusion. The researchers in the field of mobile computation often say and write that it is very difficult to give the correct definitions of a mobile agent, a mobile object, the subject of mobility. The major reason of this difficulty would be the fact that it belongs to the domain of *metaphor* by which people understand the behavior of the entire system and hence that it is a kind of a problem not to be examined by the scientific procedure. Let us take an example of such metaphor. In the Telescript computation model [41], when an agent migrate to another host, the data structure constituting the agent at the *departure* host will be completely removed after the migration. In another computation model, the data structure constituting an agent at the departure host will be also removed after the migration, but a so-called "secretary" for the agent will be left there as a proxy for the absent agent. A secretary transmits a request sent to the secretary (because the sender thinks the agent is still there) to the agent at a different location. Which computation model is better in the scientific sense?

The metaphorical problem, however, should not be depreciated because it is *on this very point* that mobile computation is distinguished from distributed computation. Mobile computation is not anything competitive to distributed computation, but is something *on it*. From the viewpoint of computationability, any mobile application program can be encoded in terms of distributed computational model. In fact, almost all available mobile systems are implemented on the existing ordinary infrastructures. They work on Solaris operating system or on Windows-NT. They utilize ordinary TCP/IP to transmit code, data and execution states which constitute mobile agents. Mobile computation is a discourse upon some metaphor, some symbolic comprehension constructed in distributed computation model. It is thus worth while investigating metaphors or concepts adopted in mobile systems from this perspective.

There is, however, a pitfall in analyzing the metaphor used in mobile computation. At the first sight, there is a variety of metaphors working in mobile systems that have been already implemented or proposed so far. Each mobile system usually has its own metaphor as what represents the distinctive features of the mobile system. But in contrast to the variety, the vigor of their discussions, it is difficult for the author to avoid being received a monotonous impression from them. In fact, the origins of the metaphors they are using in their mobile systems can be traced back to those used in Telescript. In Telescript, many aspects of mobile agents were well-considered. For instance, the programmer can specify very fine control on security such as authentication, restriction on taking objects out of a place, maximum time that an agent can live, and whether an agent can spawn a thread. The only drawback of Telescript was its extremely slow execution speed. Fortunately or unfortunately, Java does not provide such a fine control for mobile agents. So, for those who are familiar with Java and at the same time who does not know Telescript well, it is a natural consequence that many novel features that they thought and proposed is actually rediscovery of those in Telescript.

Then, is the analysis on the metaphors working in mobile systems — as expected — fruitless? Our direct answer to the question is, unfortunately, "yes", but the point not to be missed here is that although the metaphors working in mobile systems, the manifest explanations of the mobile systems are indeed monotonic, *the language mechanisms they presuppose tacitly are quite different*. When one is implementing

a new virtual machine to embody some metaphor, he can usually design and decide architecture of the virtual machine of his own free will, apart from severe realistic requirements such as execution performance and real-time responsibility. The difference among mobile systems at the architecture level thus tends to be obscure. Let us paraphrase this disguise, how a presupposed language mechanism is hidden behind a metaphor by taking an instance of the notion of "secretary" again. Suppose that an agent is referred to by remote references from external agents. When the agent migrates to another host, the programmer expects that the remote references that refers to the agent are also redirected according to the migration. But it may raise considerable costs since it requires scanning of all the remote references in the whole distributed systems. This cost increases seriously if the mobile system is working on the Internet because the network latency tends to be quite low. The effect of a secretary is to avoid this cost by omitting the remote reference update. The remote references pointing to an agent still point to the secretary after the agent migrates, and the secretary transmits a received message to the migrated agent. In contrast, Obliq adopts the rewriting of remote references, which is provided by a primitive operation `redirect`. What the notion of a "secretary" really poses is thus the issue of redirection of remote references. Recently, it has been shown that modeling location-independence of message transmission requires a complicated mechanism [86], which presents an encoding that realizes location-independent view for a mobile agent on a formal framework. This is thus a crucial implementational issue, which is undoubtedly involved in subjects of computer science.

In fact, a language mechanism that implements a programming metaphor, varies widely according to mobile systems as shown through this thesis. This thesis gives a formal framework in which various language mechanisms for mobile programming language systems can be described and compared. Furthermore, our aim includes design of a flexible mobile programming language and its efficient implementation based on the analysis.

1.2 A Remark on our Methodology to Investigate Mobile Computation

It may be somewhat misleading to say that mobile systems are analyzed formally because it reminds us of the approaches taken in the fields of artificial intelligence and software engineering. In the context of artificial intelligence [43], an agent is defined in a logical system like the Z specification language [88], and deducible properties of agents are discussed. For instance, a task that is formulated as a logical proposition is allocated to an agent. Then the agent plans to accomplish the task autonomously (or it may cooperate with the other agents), and carries out the plan. Our thesis is irrelevant to such a study.

The reason why we must avoid the discussions held in the context of software engineering such as Refs. [9, 10, 22, 26, 33, 47] is that programming metaphor itself is the subject of the discussion there. We must analyze mobile language systems not in the level of metaphor, but in the level of its presupposed language mechanism.

Then, what about concurrent process calculi? We think that it is one of the appropriate approaches on which mobile computation is studied, but it had once a tradition of way of thinking that the lack of the notion of location would be rather favorable. A typical instance of such a model is Linda's tuple space [20].

Because it lacks the notion of location completely, the programmer can forget the implementational details of distributed computation, and can concentrate upon its meaning, the behaviour of a program. It is convenient if local and remote access is transparent. Location-independence was believed as a favorable property [98].

In contrast, the notion of location must be incorporated in a calculus or a computational model in case of mobile computation since it is inevitable to express mobility. Mobile languages encourage a programming style in which code, data and execution environments are dynamically moved. Behind this paradigm, there is a presupposition that *location-dependent* operations and services should be positively exploited. This way of thinking is against the tendency in studies of distributed systems The recent development of mobile language systems is proving that there are interesting applications for such a programming style. For instance, consider an example of a complex database query. When a user of a database wants to make a complex query to a remote database, the query tends to require a series of roundtrips of a simple query. The user is forced to consume considerable time to get the final answer and the most of the waiting time is the network latency. This is a typical instance appeared in mobile computation. A program that performes the complex query at the *server site* can reduce the waiting time because a series of the intermediate data transmissions is omitted. The location of control and data is thus important to characterize this benefit.

Let us consider the problem of value transmission, that is location change of location-dependent stuff, on migration. Kali-Scheme [21] duplicates all the values reachable from a target value in principle, whereas Obliq [16] duplicates partly and the values that are not duplicated are referred to by remote references. The distinction of the ways of value transmission in these languages is important because the resulting locations of values are not identical. From our experience, the semantics of value transmission is quite different among existing mobile languages. This problem of data location must be expressed in a model of mobile computation.

Let us take an example of how the mobile point of view is lost in the traditional framework. The Java virtual machine has an on-demand class loading mechanism. A program written in Java can start its execution if all the class files constituting the program is not loaded to the runtime system. (Strictly speaking, this depends on the implementation of a Java virtual machine.) An unloaded class file is loaded on demand when it is actually needed (for instance, when an instance of the class is generated). The important point from the view of mobile computation is that the class file is loaded dynamically *on demand*. The semantics of a Java program is, however, defined not depending on its location: a Java applet runs similarly if it is executed on a different host. The semantics of the Java language does not care about when a class file is loaded. In other words, the semantics of the Java language *misses* the very point of the Java class loading mechanism from the viewpoint of mobile computation.

The recent attempts to characterize mobile computation are based on an approach to attach the notion of location to a computation model. An attempt to incorporate locality to Linda has been made [74], where *multiple* tuple spaces are provided. Many models adopt hierarchical, tree-like structures to represent locations of code and data. Fournet [36] proposes a calculus that is an extension of join calculus [37] and that can describe distributed computation and dynamic code movement. A distinctive point of the calculus is that each process has its own location and that a host (the subject of computation in a location) may be down: all the processes

in a failed host are inactive to incoming messages. The perspective in which distributed computation is characterized by *host-failure* is common to Amadio's view [5, 6, 7]. Ambient calculus [19] is an elegant process calculus, which neatly captures a notion of administrative domains and the movement of processes. Ambient calculus is suitable for expressing hierarchical structure of locations and it has several location movement operations. Distributed π-calculus [85] has hierarchical locations and mobile processes with a process movement operation, provided that the issue in [85] is to detect channels used locally. Riely [82] proposes a type system for a mobile calculus. Hennessy [51] proposes a mobile calculus in which resource access can be constrained.

The authors in Refs. [25, 42] give a different perspective to analyze mobile agent systems. They classify mobile agent systems into the following three categories from the implementational point of view.

Message-passing The capability to transmit an ordinary value (without code) to a remote computer.

Weakly mobility The capability to transmit code to a remote computer.

Strong mobility The capability to transmit code and execution states to a remote computer.

The point in their work is to describe (1) how various programming notions, such as type system, dynamic linking and security, interact with the different mobilities listed above, and (2) what kind of technologies are required to realize these mobilities. This perspective is much similar to ours in the sense that both look at mobile systems from a rather concrete, implementational viewpoint and that both classify existing mobile language systems.

1.3 Mobile Computation and the Economy of Programming

Mobile computation concerns with a certain "as if"; computation propagates to a remote computer *as if* the same program execution is resumed there. Why a certain form of distributed computation is called mobile although there are various kind of control transmission among distributed hosts? What is the form of distributed computation called mobile? First of all, mobile computation must give an illusion to the programmer as if the program execution resumed at the destination host is *as the same as* the one at the departure host. It is, however, not a trivial work because complicated mechanisms are needed to accomplish such a transparent migration [21, 83]. Then, why must such complicated mechanisms be devised? This section gives a brief justification of that illusion, an advantageous point of the mobile form of distributed computation, which constitutes a large part of our motivation to study mobile computation. Here we compare mobile computation with another programming paradigm of distributed computation called *remote evaluation* [89] in view of the economy of programming.

The go operator [45, 83, 100] is the symbolic operator of mobile computation. When a go operator is issued, the program execution continues its execution smoothly as if the operator is ignored, but the location of the program execution may be

changed to another host. The capability to transmit program execution is the distinctive feature of mobile computation from the implementational point of view [17]. In this section, we denote an idealized go operator by the form:

$$(\mathsf{go}\ p)$$

where p denotes the destination of migration.

Stamos proposes a remote evaluation scheme called REV mechanism [89], which is a mechanism to execute code at a remote computer. An REV request is written as:

$$y := \mathsf{at}\ p\ \mathsf{eval}\ f(x).$$

This form requests a function f to be applied to the value v of a variable x on a computer at location p. When the execution of $f(v)$ is finished, the result will be sent back to the requester and it will be assigned to variable y. The REV mechanism assumes that f is a closed expression and v is completely duplicated on a remote computer, which implies that the code fragments that may be sent to a remote computer can be determined at a compile time.

The REV mechanism is motivated to argue against the understanding that there is no appropriate set of more primitive operations than RPC (remote procedure calls) that describes efficiently *any* distributed application [89]. (It is surprising that the imaginary opponent of Stamos is supposed to believe that any distributed application can be described efficiently by means of RPC.) Consequently, a mechanism in which a *general* expression can be executed at a remote computer was investigated as an extension of RPC.

There are several variants of remote evaluation. In Kali-Scheme [21], a remote evaluation request is expressed as follows:

$$(\mathsf{remote\text{-}run!}\ p\ e).$$

In contrast to REV, remote-run! returns no value or void immediately. The expression e will be evaluated by another computer at location p asynchronously. If the programmer wants to receive the result of remote evaluation, he must use a communication channel explicitly. We call this form of remote evaluation *asynchronous remote evaluation* while the one in REV style is called *synchronous remote evaluation*. Kali-Scheme is an implementation of Scheme language extended with facilities for distributed computation. It has a communication channel, along which any kind of values in Scheme including continuations can be transmitted. Again unlike REV, remote-run! can have free variables, which allows dynamically determined code to be transmitted.

An inconvenience of the remote evaluation style in the practical situation arises when the programmer wants to control the location of the continuation. Such a situation often occurs in case of recursive functions and loops. Let us examine it by a simple example of tree-summation algorithm. We assume that several numbers of computers are logically connected as a binary tree. Each computer at a leaf node has an integer and a program computes its summation by traversing them. This kind of mobility may often be used in a world wide web (WWW) search robot, which gathers WWW page texts by following the links described in them. A program in the go style can be written in the Scheme-like syntax as follows:

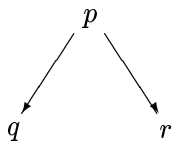
```
(define (tree-sum tree)
  (go (loc-of tree))
  (if (leaf? tree)
      (val-of tree)
      (+ (tree-sum (left-of tree)) (tree-sum (right-of tree))))).
```

where `(loc-of tree)` extracts the location of the node referred to by `tree`, `(leaf? tree)` examines whether `tree` is a leaf node, and `(val-of tree)` returns the value of the node if the program resides at a leaf node. `(left-of tree)` and `(right-of tree)` returns the left node and the right node of `tree`, respectively. We do not care how to obtain the the final answer for the programmer in this particular example.

A straightforward modification of the above program into the style of remote evaluation, however, does not preserve the pattern of control transfer among computers. Consider the following program:

```
(define (tree-sum tree)
  (if (leaf? tree)
      (val-of tree)
      (+ (at (loc-of (left-of tree)) eval (left-of tree))
         (at (loc-of (right-of tree)) eval (right-of tree)))))
```

Unfortunately, this program behaves differently from the original program. Assume that there are three locations p, q and r. Node p has node q as the left tree and node r as the right tree, which is depicted as follows:



The original program migrates from p to q, then q to r. In the modified program, the control moves along $p \to q \to p \to r \to p$. The addition takes place at location r in the original program, while it takes place at location p in the modified program. In order to preserve the behavior of the original program, we should have modified as follows:

```
(define (tree-sum tree cont)
  (remote-run! (loc-of tree)
    (if (leaf? tree)
        (cont (val-of tree))
        (tree-sum (left-of tree)
                  (lambda (left-val)
                    (tree-sum (right-of tree)
                              (lambda (right-val)
                                (+ left-val right-val))))))))
```

The function is transformed to have the second argument `cont`, which denotes the continuation of this function. This transformation looks like the continuation passing style transformation [8, 60]. Also, we must adopt asynchronous remote evaluation,

otherwise the program performs more migration since synchronous remote evaluation always returns a value.

This experience suggests that when a programmer rewrites an existing application program to be able to benefit from code migration, rewriting into the remote evaluation style may require non-trivial modification. On the other hand, rewriting into the go style requires only a minor modification at least on this particular example.

1.4 An Overview and Contributions of This Thesis

The aim of this thesis is thus threefold:

- To analyze mobile languages, a common semantic framework is provided in which various features for mobile computation in existing mobile languages can be described. Our intent on the design of the calculus is that it aims to *describe* mobile computation and at the same time to be useful for *designing* mobile languages. The mobility of a program execution is divided into the control part and the data part. Regarding the mobility of control, we found the strong similarity between the operators for migration and those for partial continuations. Regarding the mobility of data, we propose so-called data movement protocols, which can control the transparency of data migration flexibly. Some of the features of existing mobile languages are described in the calculus. This is developed in Chapter 2.

- Based on the analysis of mobile languages, a flexible programming language for mobile computation is designed. The language is an extension of an object-oriented language, such as C++, SmallTalk and Java. Surprisingly, the modification to the original language is small; nevertheless they can describe various form of migration including transparent one. The programming language is described in Chapter 3.

- An implementation of the programming language is constructed to show the advantages of the programming language. We design and implement a mobile programming language as an extension of the Java language. The implementation is so portable that the program runs on any standard Java interpreter and can be compiled by any standard Java just-in-time (JIT) compiler. A program in the mobile language runs as fast as the non-mobile execution of the program. This is presented in Chapter 4.

1.5 The Definitions of Mathematical Symbols and Notations

This section gives the notational definitions used throughout the thesis.

$\equiv$ The syntactic equivalence. $x \equiv y$ denotes the fact that x and y are syntactically equivalent.

$=_\alpha$ $e =_\alpha e'$ denotes the fact that two lambda terms e and e' are α-equivalent with each other.

$=_{\alpha\beta}$ $e =_{\alpha\beta} e'$ denotes the fact that two lambda terms e and e' are $\alpha\beta$-equivalent with each other.

$=_{\alpha\beta\eta}$ $e =_{\alpha\beta\eta} e'$ denotes the fact that two lambda terms e and e' are $\alpha\beta\eta$-equivalent with each other.

$\rightarrow_{\alpha\beta}$ $e \rightarrow_{\alpha\beta} e'$ denotes the fact that lambda term e can be reducible to e' by using either α- or β-reduction.

$\rightarrow^*_{\alpha\beta}$ $\rightarrow^*_{\alpha\beta}$ is the reflexive and transitive closure of $\rightarrow_{\alpha\beta}$.

Dom(f), **Rng**(f) Let f be a function. Then Dom(f) denotes the domain of f, while Rng(f) denotes the range (codomain) of f.

$\{x_1 \mapsto y_1, ..., x_n \mapsto y_n\}$ A finite function that maps each x_i with y_i for $i = 1..n$.

$\uplus$ The disjoint union. $x \uplus y$ is equivalent to $x \cup y$ provided that x and y must be disjoint.

$\setminus$ The set subtraction. $x \setminus y = \{z | z \in x, z \notin y\}$.

$f.\{x \mapsto y\}$ When f is a finite map, $f.\{x \mapsto y\}$ denotes the finite map equivalent to f except that $(f.\{x \mapsto y\})(x)$ is defined to be y.

$f.\{x_1 \mapsto y_1, ..., x_n \mapsto y_n\}$ An abbreviation of $(f.\{x_1 \mapsto y_1\}...).\{x_n \mapsto y_n\}$.

$[y_1/x_1, ..., y_n/x_n]$ A substitution that replaces each occurrence of x_i with y_i for $i = 1..n$.

θ_x When θ is a substitution, θ_x denotes the substitution equivalent to θ except that $\theta_x(x) = x$.

$\emptyset$ The empty set. The empty map. The empty substitution.

ε The empty sequence.

$|\alpha|$ Let α be a sequence. The $|\alpha|$ denotes the length of α.

$f \sqsubseteq g$ Let f, g be functions. We say that g is an extension of f if Dom$(f) \subseteq$ Dom(g) and $f(x) = g(x)$ for any $x \in$ Dom(f). The relation $\sqsubseteq$ is reflexive and transitive.

$H \sqsubseteq H'$ In case that H, H' are heaps (defined in Section 2.2), the extension relation is defined as follows. We say that H' is an extension of H if Dom$(H) \subseteq$ Dom(H'), $H(a) = (P, v)$, $H'(a) = (P', v')$ and $v = v'$ for any $a \in$ Dom(H). The relation is also reflexive and transitive.

$\mathbb{H} \sqsubseteq \mathbb{H}'$ In case that H, H' are global heaps (defined in Section 2.2), the extension relation is defined as point-wise extension of the extension relation between heaps. That is, we say $\mathbb{H} \sqsubseteq \mathbb{H}'$ if $\mathbb{H}(p) \sqsubseteq \mathbb{H}'(p)$ for any $p \in$ Dom$(\mathbb{H})$. The relation is also reflexive and transitive.

θf Let θ be a substitution and f be a finite mapping $\{x_1 \mapsto y_1, ..., x_n \mapsto y_n\}$. Then the composition θf is defined to be a finite mapping $\{x_1 \mapsto \theta y_1, ..., x_n \mapsto \theta y_n\}$.

Chapter 2

A Calculus with Code Mobility

Distributed programming language systems in which code, data and/or execution states move around among computer systems [1, 16, 23, 44, 45, 62, 68, 92, 94, 101] have been proposed with the development of telecommunication networks. These systems are usually equipped with their own features that are hard to simulate by other systems even with respect to their agent movement (migration) mechanisms. The difference is due to the different presuppositions on application, security, availability, heterogeneity, efficiency and so on. In order to compare and investigate the difference among agent systems, we have to put them on a common formal foundation in which their agent movement mechanisms can be described. In this chapter, we describe a simple and flexible calculus, which provides a neat tool for describing transference mechanisms of code, data and execution states. The distinctive feature of our calculus is its capability of representing heap modification and control delivery on migration. Since the major difference in existing mobile languages appears in the way to handle heaps on migration, the capability is important. We demonstrate that the mechanisms for mobility in Obliq [16], Telescript [101], Kali-Scheme [21] and Java [44] can be described.

2.1 Calculus Overview

The most distinguished points of mobile agent systems from other language systems is their facility to move code and execution state to another site *dynamically*. In this section, we describe how to represent the mechanism, dynamic transference of code and execution state in the source language. Then, one of the fundamental notions concerning to data mobility, called *data movement protocols*, is described.

2.1.1 Code Mobility

We use an extension of call-by-value λ-calculus to express code, which is defined by the following grammar:

$$e ::= p \mid x \mid (\lambda x.e) \mid (ee) \mid (\mathsf{go}\ e) \mid \langle e \rangle.$$

where x is a variable, $(\lambda x.e)$ is an abstraction and (ee) is an application. We use p to denote a *place* or a *location*. A location is a logical notion of a host or a site, which accommodates migrating programs. The presence of multiple threads in a location is possible. A go expression $(\mathsf{go}\ e)$ issues migration where the argument designates its

destination location. Finally $\langle e \rangle$ is called an *undock expression*, which denotes a unit of code movement or an agent. In addition, we often use a let expression as a syntax sugar such that $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \equiv (\lambda x.e_2)e_1$. A sequential execution $(e_1; e_2)$ is encoded as $\mathsf{let}\ x = e_2\ \mathsf{in}\ e_1$ for some $x \notin \mathrm{FV}(e_1)$. We denote by $\mathsf{let}\ x_1 = e_1, ..., x_n = e_n\ \mathsf{in}\ e$ a sequence of let-expressions $\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ ...\ \mathsf{in}\ \mathsf{let}\ x_n = e_n\ \mathsf{in}\ e$, which is similar to a so-called let* expression. The set of free variables in an expression is defined inductively as follows:

$$\begin{aligned} \mathrm{FV}(p) &= \emptyset, & \mathrm{FV}(e_1 e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2), \\ \mathrm{FV}(x) &= \{x\}, & \mathrm{FV}(\mathsf{go}\ e) &= \mathrm{FV}(e), \\ \mathrm{FV}(\lambda x.e) &= \mathrm{FV}(e) \setminus \{x\}, & \mathrm{FV}(\langle e \rangle) &= \mathrm{FV}(e). \end{aligned}$$

The code movement operation in the calculus is issued by the following go expression that is quite a similar mechanism to that of Telescript:

$$(\mathsf{go}\ p)$$

where p denotes a destination. Unlike Telescript, the argument is a simple location, not a ticket (a capability to enter the destination provided in Telescript). After evaluating a go expression, its continuation will be moved to the destination and then evaluated there.

$$\mathsf{let}\ x = 3, y = (\mathsf{go}\ p)\ \mathsf{in}\ (fx).$$

In the above code fragment, (fx) is evaluated in location p. Multiple hopping is realized just by a repetition of go expressions:

$$\mathsf{let}\ x = 3, y = (\mathsf{go}\ p), z = (\mathsf{go}\ q)\ \mathsf{in}\ (fx).$$

In this case, the expression moves to location p, then moves to location q and finally (fx) is evaluated at location q.

In all the above examples, the entire continuation on the point of a go expression always migrates to the destination. This form of migration is called *transparent migration* [40, 45].

Definition 2.1.1 (Migration Transparency). Migration is called *transparent* if the execution of a migrating program is resumed at a destination site with exactly the same execution state as that of the migration time.

Transparent migration is useful for the programmer to understand the behavior of a migrating program. It has, however, some practical drawbacks with respect to execution performance of migration:

- The entire continuation is usually a large data structure since it is almost equal to the entire program execution. Some mechanism is thus necessary to restrict the continuation to be sent [21]. In practice, it is often the case that migration of a small program satisfies the programmer's requirement.

- According to Refs. [21, 62], a large part of the data structures sent to a destination on migration is also provided in the destination host in advance (for instance, components constituting the runtime system.) Therefore the total amount of data transmitted on migration can be reduced by omitting transmission of those sharable data structures.

Because of the first point, we introduce an undock expression $\langle e \rangle$, that restricts the continuation to be sent on migration. Such a migrating part is usually called an agent or a mobile object. The second point concerns data aspects of migration, which is more discussed in the next subsection.

Definition 2.1.2 (The Migrating Code Area). When a go expression is evaluated, the area of code moved to a destination is the innermost undock expression surrounding the go expression.

$$\mathsf{let}\ x = 2, y = \langle \mathsf{let}\ z = (\mathsf{go}\ q)\ \mathsf{in}\ (g\ 3) \rangle\ \mathsf{in}\ (f\ x).$$

In the above example, the code to be moved, when $(\mathsf{go}\ q)$ is evaluated at location p, is the innermost undock expression surrounding $(\mathsf{go}\ q)$, that is $\langle \mathsf{let}\ z = (\mathsf{go}\ q)\ \mathsf{in}\ (g\ 3) \rangle$. The expression $(g\ 3)$ is evaluated at location q and $(f\ x)$ is evaluated at location p.

Now we can define asynchronous remote evaluation as a simple macro using a go and an undock expressions:

$$(\mathsf{remote\text{-}run!}\ p\ e) \equiv \langle \mathsf{let}\ d = (\mathsf{go}\ p)\ \mathsf{in}\ e \rangle = \langle \mathsf{go}\ p; e \rangle$$

for some d is not free in e. The expression in the left side migrates to p, and then e is evaluated in the destination. Since the let-expression is enclosed within an undock expression, expressions containing the remote-run! do not migrate.

2.1.2 Data Mobility

In order for code mobility to be significant, we must have a distinction between something local and something remote. If there is no difference between local evaluation and remote one, why should we move code? We thus introduce a local heap for each location to represent local resources. Any value in a local heap is accessible *only* from the code in the same place. A heap is a finite map from addresses to values, written as follows:

$$\{a \mapsto (P, 7), b \mapsto (P', \texttt{"abc"})\}.$$

We denote by the above notation a heap where integer 7 is stored at address a and string `"abc"` is stored at address b. The paired symbols P, P' with heap values are meta variables denote *data movement protocols.*

In this calculus, a data movement protocol is associated with each address and specifies the behavior of the stored value on migration. When a value is moved onto other location, it may be duplicated, become a remote reference or be nullified. Data movement mechanisms vary widely depending on mobile language systems. For instance, Telescript and Obliq have quite different data movement mechanisms. A remote value can be referred to by a remote reference in Obliq, while there is no remote reference in Telescript. In Obliq, the behavior of data movement changes according to immutability of values. When a value is immutable such as an integer and a function, it is copied to a destination. In contrast when a value is mutable like a reference cell, it is not copied but is referred to by a remote reference. The consistency of the content of a reference cell is guaranteed because it is never duplicated. In Telescript, the reachable values from a migrating agent are simply duplicated since there is no remote reference. When a migrating agent has a reference to another agent, the latter agent is not moved, but the reference to the agent is disconnected or nullified at the time of migration.

We introduce 6 kinds of data movement protocols, which are sufficient to describe movement mechanisms of almost all existing mobile language systems. We illustrate these protocols in sequence.

Copy protocol:

	before migration	after migration	translation
departure p	$\{a \mapsto (\mathbf{copy}, v), ...\}$	$\{a \mapsto (\mathbf{copy}, v), ...\}$	$\emptyset$
destination q	$\{...\}$	$\{a' \mapsto (\mathbf{copy}, v'), ...\}$	$\{a \mapsto a'\}$ (a' fresh)

This table shows how the heaps at the current place and the destination place change before and after data movement. The value v' is a duplication of v at the destination. It is stored at address a', which is fresh in the destination heap. The translation means how the moved address changes after the movement. The address translation at the current place is empty since one needs no address translation at the current place for this type. Because value v referred to by address a before the movement will be referred by address a' after the movement at the destination, we have to replace address a with address a' at the destination. This type of movement is so-called copy operation. A value v is copied to a new space a'. The duplicated value at the destination is denoted by v', not by v since they can be different. If v is also an address, the rules of data movement are applied recursively. Therefore the duplicated value v' can be a *local* reference in the destination heap.

Resident protocol:

	before migration	after migration	translation
departure p	$\{a \mapsto (\mathbf{resident}, v), ...\}$	$\{a \mapsto (\mathbf{resident}, v), ...\}$	$\emptyset$
destination q	$\{...\}$	$\{...\}$	$\{a \mapsto a@p\}$

where $a@p$ denotes a remote reference pointing to address a in location p. This 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. The remote objects and remote interfaces in JavaRMI behave according to resident protocol. Every remote object is registered in a RMI registry (an object request broker in terms of CORBA). When a RMI registry is asked a remote object, it returns a remote reference for the object. The value stored in a remote reference (that is actually a proxy object for the remote object) can be transmitted to other hosts. A remote object does not move, but the remote reference is in circulation among hosts. The address translation at the destination maps address a to remote reference $a@p$ since v is referred as a remote reference from the outside.

Carry protocol:

	before migration	after migration	translation
departure p	$\{a \mapsto (\mathbf{carry}, v), ...\}$	$\{a \mapsto (\mathbf{carry}, v), ...\}$	$\{a \mapsto a'@q\}$
destination q	$\{...\}$	$\{a' \mapsto (\mathbf{carry}, v'), ...\}$	$\{a \mapsto a'\}$ (a' fresh)

This protocol contrasts with the previous protocol. A value of carry protocol is moved to the destination and is referred to by a remote reference from the *departure*. It seems that the value migrates to the destination. The value v is duplicated to the destination, which is denoted by v'. The value v left at the departure is not referred to from the departure since all references a in the departure store are replaced with a remote reference $a'@q$ by the address translation. The address translation at the current place is thus $\{a \mapsto a'@q\}$.

Proper protocol:

	before migration	after migration	translation
departure p	$\{a \mapsto (\mathbf{proper}, v), ...\}$	$\{a \mapsto (\mathbf{proper}, v), ...\}$	$\emptyset$
destination q	$\{...\}$	$\{...\}$	$\{a \mapsto \bot\}$

where $\bot$ denotes the undefined value. This type represents a value that never goes out and that cannot be referred to from the outside, such as special system objects and highly secure values. Every address a is replaced with the undefined value $\bot$ at the destination, thus the agent cannot refer to the value after the migration. An instance of this protocol is a non-owner reference in Telescript. References are divided into owner ones and non-owner ones in Telescript. When an agent migrates, all the values referred to by owner references from the agent are moved to the destination with the agent. On the other hand, if a value is referred to by only non-owner references, it is not copied, but the references are nullified.

Takeaway protocol:

	before migration	after migration	translation
departure p	$\{a \mapsto (\mathbf{takeaway}, v), ...\}$	$\{a \mapsto (\mathbf{takeaway}, v), ...\}$	$\{a \mapsto \bot\}$
destination q	$\{...\}$	$\{a' \mapsto (\mathbf{takeaway}, v'), ...\}$	$\{a \mapsto a'\}$

In contrast to the previous case, a value of this type is considered to be inherently possessed by an agent. It looks as if a value is gone with an agent. Every address a in the departure is replaced with the undefined value $\bot$ so that anyone there cannot refer to the data after the movement.

Ubiq protocol:

	before migration	after migration	translation
departure p	$\{a \mapsto (\mathbf{ubiq}, v), ...\}$	$\{a \mapsto (\mathbf{ubiq}, v), ...\}$	$\emptyset$
destination q	$\{a \mapsto (\mathbf{ubiq}, v'), ...\}$	$\{a \mapsto (\mathbf{ubiq}, v'), ...\}$	$\emptyset$

Since values to be transmitted of the previous protocols are always allocated on a fresh area in the destination heap, they are not referred to from existing values in the destination store. Conversely, transmitted values have no reference to existing values in the destination heap if the transmitted values do not have references to them in advance. Therefore, to *connect* transmitted values with existing values in the destination heap, we need a service of dynamic linking, such as ORB (object request broker) or ubiquitous value [62, 84].

This protocol is used when both departure and destination share objects mainly for the purpose of reducing the total size of data transmission. In Kali-Scheme, the value of this protocol is designated by a unique identifier to omit transmission. In Facile [62], the value of this protocol is called a ubiquitous value. It is often the case that some kind of objects are already shared between departure and destination like objects used in the runtime system. It is wasteful to transmit such objects in view of transmission duration. Such shared objects is not necessary to transmit, but the objects at the destination is exploited.

2.2 Operational Semantics

This section gives an operational semantics of our mobile calculus.

2.2.1 Local Semantics

The semantic objects for the calculus are listed in Figure 2.1. The set *Const* of constants such as boolean truth values, character strings and integers is assumed.

$$
\begin{aligned}
c &\in \mathit{Const} \\
a &\in \mathit{Ref} \\
u &\in \mathit{UbiqRef} \\
p &\in \mathit{Loc} \\
a@p \in \mathit{RemRef} &= \mathit{Ref} \times \mathit{Loc} \\
P \in \mathit{Proto} &::= \mathbf{copy} \mid \mathbf{resident} \mid \mathbf{carry} \mid \mathbf{proper} \mid \mathbf{takeaway} \mid \mathbf{ubiq} \\
x &\in \mathit{Var} \\
v \in \mathit{Val} &::= () \mid c \mid x \mid p \mid a \mid u \mid a@p \mid P \mid \lambda x.e \\
e \in \mathit{Exp} &::= v \mid ee \mid (\mathsf{go}\ e) \mid \langle e \rangle \mid \mathsf{new}(e)e \mid\ !e \mid\ ?e \mid \mathsf{set}(e)ee \\
H \in \mathit{Heap} &::= \{a \mapsto (P, v), ..., u \mapsto (P', v'), ...\} \\
&\qquad : (\mathit{Ref} \uplus \mathit{UbiqRef}) \overset{\text{fin}}{\to} \mathit{Proto} \times \mathit{Val} \\
\mathbb{H} \in \mathit{GlobalHeap} &::= \{p_1 \mapsto H_1, ..., p_n \mapsto H_n\} : \mathit{Loc} \overset{\text{fin}}{\to} \mathit{Heap}
\end{aligned}
$$

Figure 2.1: Domains.

We denote by () a special constant called the *unit*. The set of references consist of usual (local) references a and ubiquitous ones u. A remote reference is a pair of a local reference and a location. A ubiquitous reference does not constitute a remote reference since it is supposed to be ubiquitous (that is each heap has its value). The expressions are lambda expressions extended by a go expression, an undock expression and operations on a heap. There are four heap operations (new, !, ? and set); an initialization $\mathsf{new}(e)e'$ creates a new reference pointing to e' of data movement protocol e. A value dereference $!e$ fetches the value stored at address e. A protocol dereference $?e$ fetches the data movement protocol of address e. A memory update $\mathsf{set}(e)e'e''$ updates the content of address e' with new value e'' and new protocol e. A heap is a finite map from a reference to a pair of a data movement protocol and a value. $\lambda x.\lambda y.e$ can be abbreviated as $\lambda xy.e$. Application is right-associative, that is $e_1e_2e_3 \equiv (e_1e_2)e_3$.

The set of free variables in an expression is defined by the following function FV:

$$
\begin{aligned}
\mathrm{FV}(()) &= \emptyset & \mathrm{FV}(\lambda x.e) &= \mathrm{FV}(e) \setminus \{x\} \\
\mathrm{FV}(c) &= \emptyset & \mathrm{FV}(ee') &= \mathrm{FV}(e) \cup \mathrm{FV}(e') \\
\mathrm{FV}(x) &= \{x\} & \mathrm{FV}(\mathsf{go}\ e) &= \mathrm{FV}(e) \\
\mathrm{FV}(p) &= \emptyset & \mathrm{FV}(\mathsf{new}(e)e') &= \mathrm{FV}(e) \cup \mathrm{FV}(e') \\
\mathrm{FV}(a) &= \emptyset & \mathrm{FV}(!e) &= \mathrm{FV}(e) \\
\mathrm{FV}(u) &= \emptyset & \mathrm{FV}(?e) &= \mathrm{FV}(e) \\
\mathrm{FV}(a@p) &= \emptyset & \mathrm{FV}(\mathsf{set}(e)e'e'') &= \mathrm{FV}(e) \cup \mathrm{FV}(e') \cup \mathrm{FV}(e'') \\
\mathrm{FV}(P) &= \emptyset
\end{aligned}
$$

Besides, the set of all *local* addresses in e is denoted by $\mathrm{LA}(e)$.

$$
\begin{aligned}
\mathrm{LA}(()) &= \emptyset & \mathrm{LA}(\lambda x.e) &= \mathrm{LA}(e) \\
\mathrm{LA}(c) &= \emptyset & \mathrm{LA}(ee') &= \mathrm{LA}(e) \cup \mathrm{LA}(e')
\end{aligned}
$$

$$\begin{aligned}
\mathrm{LA}(x) &= \emptyset & \mathrm{LA}(\mathsf{go}\ e) &= \mathrm{LA}(e)\\
\mathrm{LA}(p) &= \emptyset & \mathrm{LA}(\mathsf{new}(e)e') &= \mathrm{LA}(e) \cup \mathrm{LA}(e')\\
\mathrm{LA}(a) &= \{a\} & \mathrm{LA}(!e) &= \mathrm{LA}(e)\\
\mathrm{LA}(u) &= \emptyset & \mathrm{LA}(?e) &= \mathrm{LA}(e)\\
\mathrm{LA}(a@p) &= \emptyset & \mathrm{LA}(\mathsf{set}(e)e'e'') &= \mathrm{LA}(e) \cup \mathrm{LA}(e') \cup \mathrm{LA}(e'')\\
\mathrm{LA}(P) &= \emptyset
\end{aligned}$$

The set of all *remote* addresses in e is denoted by $\mathrm{RA}(e)$.

$$\begin{aligned}
\mathrm{RA}(()) &= \emptyset & \mathrm{RA}(\lambda x.e) &= \mathrm{RA}(e)\\
\mathrm{RA}(c) &= \emptyset & \mathrm{RA}(ee') &= \mathrm{RA}(e) \cup \mathrm{RA}(e')\\
\mathrm{RA}(x) &= \emptyset & \mathrm{RA}(\mathsf{go}\ e) &= \mathrm{RA}(e)\\
\mathrm{RA}(p) &= \emptyset & \mathrm{RA}(\mathsf{new}(e)e') &= \mathrm{RA}(e) \cup \mathrm{RA}(e')\\
\mathrm{RA}(a) &= \emptyset & \mathrm{RA}(!e) &= \mathrm{RA}(e)\\
\mathrm{RA}(u) &= \emptyset & \mathrm{RA}(?e) &= \mathrm{RA}(e)\\
\mathrm{RA}(a@p) &= \{a@p\} & \mathrm{RA}(\mathsf{set}(e)e'e'') &= \mathrm{RA}(e) \cup \mathrm{RA}(e') \cup \mathrm{RA}(e'')\\
\mathrm{RA}(P) &= \emptyset
\end{aligned}$$

The operational semantics for the local, sequential evaluation is defined by a rewriting system using evaluation contexts. Given a heap H, the operational semantics associates an expression e with the reduced expression e' and the possibly updated heap H'. This is denoted by the form $\mathsf{letrec}\ H\ \mathsf{in}\ e \rightsquigarrow \mathsf{letrec}\ H'\ \mathsf{in}\ e'$. The evaluation context is an expression that contains only one special constant $[\,]$ called a *hole*. It is defined by the following grammar:

$$\begin{aligned}
E ::= & [\,] \mid Ee \mid vE \mid (\mathsf{go}\ E) \mid \langle E \rangle \mid \mathsf{new}(E)e \mid \mathsf{new}(P)E \mid !E \mid ?E\\
& \mid \mathsf{set}(E)ee \mid \mathsf{set}(P)Ee \mid \mathsf{set}(P)vE.
\end{aligned}$$

Let E be an evaluation context, we denote by $E[e]$ an expression syntactically equivalent to E except that the hole in E is replaced with e. For instance, let E be context $(\lambda x.x)[\,]$, then $E[3]$ is expression $(\lambda x.x)3$. The evaluation contexts E are used to determine the redex of an expression. The hole in a context designates precisely its current evaluation point by the call-by-value evaluation strategy. If e' is the redex of e, then there exists a context E such that $e \equiv E[e']$.

We define another kind of evaluation contexts for defining the code area to be moved on migration. An evaluation context is called *undock-free* if the hole in the context is not enclosed within undock expressions. The undock-free evaluation context is constructed by the following grammar:

$$\begin{aligned}
A ::= & [\,] \mid Ae \mid vA \mid (\mathsf{go}\ A) \mid \mathsf{new}(A)e \mid \mathsf{new}(P)A \mid !A \mid ?A\\
& \mid \mathsf{set}(A)ee \mid \mathsf{set}(P)Ae \mid \mathsf{set}(P)vA.
\end{aligned}$$

The difference on the grammar from that of the ordinary evaluation context is that it lacks the rule of the form $\langle A \rangle$. The hole in an undock-free evaluation context thus cannot be enclosed within undock expressions. Undock expressions can appear in an undock-free evaluation context if they does not enclose the hole. For instance, $([\,]\langle \lambda x.x \rangle)$ is a valid undock-free evaluation context.

$$\mathsf{letrec}\ H\ \mathsf{in}\ E[(\lambda x.e)v] \rightsquigarrow \mathsf{letrec}\ H\ \mathsf{in}\ E[[v/x]e] \quad \text{(App)}$$

$$\mathsf{letrec}\ H\ \mathsf{in}\ E[\langle v\rangle] \rightsquigarrow \mathsf{letrec}\ H\ \mathsf{in}\ E[v] \quad \text{(Undock)}$$

$$\mathsf{letrec}\ H\ \mathsf{in}\ E[\mathsf{new}(P)v] \rightsquigarrow \mathsf{letrec}\ H \uplus \{a \mapsto (P,v)\}\ \mathsf{in}\ E[a] \quad \text{(New)}$$

$$\mathsf{letrec}\ \{..., a \mapsto (P,v), ...\}\ \mathsf{in}\ E[!a] \rightsquigarrow \mathsf{letrec}\ \{..., a \mapsto (P,v), ...\}\ \mathsf{in}\ E[v] \quad (!)$$

$$\mathsf{letrec}\ \{..., a \mapsto (P,v), ...\}\ \mathsf{in}\ E[?a] \rightsquigarrow \mathsf{letrec}\ \{..., a \mapsto (P,v), ...\}\ \mathsf{in}\ E[P] \quad (?)$$

$$\mathsf{letrec}\ \{..., a \mapsto (P,v), ...\}\ \mathsf{in}\ E[\mathsf{set}(P')av'] \rightsquigarrow \mathsf{letrec}\ \{..., a \mapsto (P',v'), ...\}\ \mathsf{in}\ E[()] \quad \text{(Set)}$$

$$\mathsf{letrec}\ \{..., u \mapsto (P,v), ...\}\ \mathsf{in}\ E[!u] \rightsquigarrow \mathsf{letrec}\ \{..., u \mapsto (P,v), ...\}\ \mathsf{in}\ E[v] \quad (!u)$$

$$\mathsf{letrec}\ \{..., u \mapsto (P,v), ...\}\ \mathsf{in}\ E[?u] \rightsquigarrow \mathsf{letrec}\ \{..., u \mapsto (P,v), ...\}\ \mathsf{in}\ E[P] \quad (?u)$$

Figure 2.2: The operational semantics for local evaluation.

The substitution on a context is defined as follows:

$$\theta[\,] = [\,] \qquad \theta P = P$$

$$\theta() = () \qquad \theta(\lambda x.e) = \lambda x.\theta_x e$$

$$\theta c = c \qquad \theta(ee') = (\theta e)(\theta e')$$

$$\theta x = \begin{cases} y & \text{if } y/x \in \theta \\ x & \text{if } x \notin \mathrm{Dom}(\theta) \end{cases} \qquad \theta(\mathsf{go}\ e) = (\mathsf{go}(\theta e))$$

$$\theta(\mathsf{new}(e)e') = \mathsf{new}(\theta e)(\theta e')$$

$$\theta p = p \qquad \theta(!e) = !(\theta e)$$

$$\theta a = \begin{cases} y & \text{if } y/a \in \theta \\ a & \text{if } a \notin \mathrm{Dom}(\theta) \end{cases} \qquad \theta(?e) = ?(\theta e)$$

$$\theta(\mathsf{set}(e)e'e'') = \mathsf{set}(\theta e)(\theta e')(\theta e'')$$

$$\theta u = u$$

$$\theta(a@p) = \begin{cases} y & \text{if } y/a@p \in \theta \\ a@p & \text{if } a@p \notin \mathrm{Dom}(\theta) \end{cases}$$

Here we use only substitutions that replace variables, local references, and remote references. Note that remote references can be substituted though they are not atomic.

The rewriting rules for the local operational semantics is listed in Figure 2.2. The rewriting rule for an undock expression shows that an undock expression is just ignored if no migration takes place in it. There are no rules to update the contents of ubiquitous values because we intend ubiquitous values to represent system runtime objects. The semantics of go expressions is not defined here because it requires distributed, non-local environments.

2.2.2 Formulation of Data Transfer

During migration, values on the heap are duplicated to the destination. Here we formulate the data transfer on migration. The formulation of data transfer is quite important because:

1. The difference among existing mobile language systems is heavily due to the point of data transfer, and

2. The formulation of data transfer is weakly supported in many mobile calculi because they have no heap or store.

During data transfer, the following considerations must be taken:

Allocation of new memory The data being transmitted are usually stored on a memory area allocated newly for the migration at the destination. When the data has an internal reference to a value within the data, the value will be placed in a newly allocated memory after the migration. The reference thus must also be replaced with the one within the newly allocated memory.

Sharing When the data being transmitted have internal references pointing to the same value within the data, these references must be replaced in a consistent manner. Otherwise, sharing will be violated.

Pointer forwarding The values not directly involved to migration are sometimes affected by migration. Consider the carry protocol. When a migrating program carries some value to the destination, the references pointing to the value in the departure will be forwarded so that they will point to the new location of the value. This operation is so-called pointer forwarding. The data transfer of takeaway protocol is also considered as pointer forwarding. A reference of takeaway protocol is nullified after data transfer in the departure. It is considered as pointer forwarding to the null value.

These requirements are embodied in the rules of data transfer, called *data movement relations*, listed in Figure 2.3 and Figure 2.4. These rules are formulated as a relation between *effects* of data transfer. An effect is a quadruple (σ, Δ, H, v) where:

σ : the pointer forwarding on the departure,
Δ : the mapping from the references in the departure to the fresh ones in the destination,
H : the newly allocated heap space on the destination, and
v : the moving value.

By using the effects, a data movement relation is writen generally of the form:

$$\sigma, \Delta, H_d, v \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, v'.$$

where p is the departure location, p' is the destination location, H is the departure heap, H_d is the destination heap before moving v and H'_d is the destination heap after moving v. The lefthand side expresses the effect before moving v and the righthand side expresses the effect after the movement. Intuitively, this relation states that when value v is moved, it becomes v' in the destination, the destination heap is extended with H'_d, and pointer forwarding σ' must be applied to the departure heap. The rules in Figure 2.3 is the rules for address transfer, while the rules in Figure 2.4 is for the other kind of values. We assume that (1) $\mathrm{Dom}(\Delta) \subseteq \mathrm{Dom}(H)$ and (2) $\mathrm{Rng}(\Delta) = \mathrm{Dom}(H_d)$ because (1) the data to be moved must be in the departure heap and (2) the transferred data must be in the destination heap.

Pointer forwarding is represented by σ. When $(a, a') \in \sigma$, it means that the reference a is forwarded so that it refers to the reference a' in the *destination* heap on migration, that is a is replaced with $a'@p'$. This operation is denoted by σH,

$$\frac{H(a) = (\mathbf{copy}, v) \qquad \sigma, \Delta \uplus \{(a, a')\}, H_d \uplus \{a' \mapsto (\mathbf{copy}, v')\}, v \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, v'}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, a'} \text{ (Move Copy)}$$

$$\frac{H(a) = (\mathbf{resident}, v)}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, a@p} \text{ (Move Resident)}$$

$$\frac{H(a) = (\mathbf{carry}, v) \qquad \sigma \uplus \{(a, a')\}, \Delta \uplus \{(a, a')\}, H_d \uplus \{a' \mapsto (\mathbf{carry}, v')\}, v \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, v'}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, a'} \text{ (Move Carry)}$$

$$\frac{H(a) = (\mathbf{proper}, v)}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, \bot} \text{ (Move Proper)}$$

$$\frac{H(a) = (\mathbf{takeaway}, v) \qquad \sigma \uplus \{(a, \bot)\}, \Delta \uplus \{(a, a')\}, H_d \uplus \{a' \mapsto (\mathbf{takeaway}, v')\}, v \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, v'}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, a'} \text{ (Move Takeaway)}$$

$$\frac{}{\sigma, \Delta, H_d, u \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, u} \text{ (Move Ubiq)}$$

$$\frac{}{\sigma, \Delta, H_d, a@p' \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, a} \text{ (Move Ref1)}$$

$$\frac{p'' \neq p'}{\sigma, \Delta, H_d, a@p'' \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, a@p''} \text{ (Move Ref2)}$$

$$\frac{(a, a') \in \Delta}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, a'} \text{ (Move Addr)}$$

Figure 2.3: The rules for address transfer.

$$\frac{k \in \{(), c, x, p, P\}}{\sigma, \Delta, H_d, k \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, k} \text{ (Move Const)}$$

$$\frac{\sigma, \Delta, H_d, e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'}{\sigma, \Delta, H_d, \lambda x.e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, \lambda x.e'} \text{ (Move Abs)}$$

$$\frac{\sigma, \Delta, H_d, e_1 \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'_1 \quad \sigma', \Delta', H'_d, e_2 \Rightarrow_{H,p,p'} \sigma'', \Delta'', H''_d, e'_2}{\sigma, \Delta, H_d, e_1 e_2 \Rightarrow_{H,p,p'} \sigma'', \Delta'', H''_d, e'_1 e'_2} \text{ (Move App)}$$

$$\frac{\sigma, \Delta, H_d, e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'}{\sigma, \Delta, H_d, (\mathsf{go}\ e) \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, (\mathsf{go}\ e')} \text{ (Move Go)}$$

$$\frac{\sigma, \Delta, H_d, e_1 \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'_1 \quad \sigma', \Delta', H'_d, e_2 \Rightarrow_{H,p,p'} \sigma'', \Delta'', H''_d, e'_2}{\sigma, \Delta, H_d, \mathsf{new}(e_1) e_2 \Rightarrow_{H,p,p'} \sigma'', \Delta'', H''_d, \mathsf{new}(e'_1) e'_2} \text{ (Move New)}$$

$$\frac{\sigma, \Delta, H_d, e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'}{\sigma, \Delta, H_d, !e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, !e'} \text{ (Move !)}$$

$$\frac{\sigma, \Delta, H_d, e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'}{\sigma, \Delta, H_d, ?e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, ?e'} \text{ (Move ?)}$$

$$\frac{\begin{array}{c}\sigma, \Delta, H_d, e_1 \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'_1 \\ \sigma', \Delta', H'_d, e_2 \Rightarrow_{H,p,p'} \sigma'', \Delta'', H''_d, e'_2 \\ \sigma'', \Delta'', H''_d, e_3 \Rightarrow_{H,p,p'} \sigma''', \Delta''', H'''_d, e'_3\end{array}}{\sigma, \Delta, H_d, \mathsf{set}(e_1) e_2 e_3 \Rightarrow_{H,p,p'} \sigma'', \Delta'', H''_d, \mathsf{set}(e'_1) e'_2 e'_3} \text{ (Move Set)}$$

Figure 2.4: The rules for data transfer.

which denotes the new heap where all the references to be forwarded are forwarded. The referent of a forwarded pointer must have been duplicated in the destination heap. This requirement is represented by $\sigma \subseteq \Delta$. We assume that $\sigma \subseteq \Delta$ when the data movement relation is presented.

2.2.3 Formulation of Distributed Computation

We have never mentioned so far about concurrency of distributed computation. But there are a lot of locations in the world and each location may have several activities. We must thus perform *multiple* evaluations at the same time. The concurrent computation in our calculus is represented by a standard interleaving semantics. It is formulated as a transition relation $\rightarrow$ between global states. A global state consists of a pair of a global store $\mathbb{H}$ and a finite set $\mathbb{E}$ of pairs of locations and expressions.

$$(\mathbb{H}, \mathbb{E}) \rightarrow (\mathbb{H}', \mathbb{E}').$$

The relation between local evaluation and distributed computation is established by the following rule:

$$\frac{(p, e) \in \mathbb{E} \qquad \mathsf{letrec}\ \mathbb{H}(p)\ \mathsf{in}\ e \rightsquigarrow \mathsf{letrec}\ H'\ \mathsf{in}\ e' \qquad \mathbb{H}' = \mathbb{H}.\{p \mapsto H'\} \qquad \mathbb{E}' = \mathbb{E} \setminus (p, e) \cup \{(p, e')\}}{(\mathbb{H}, \mathbb{E}) \rightarrow (\mathbb{H}', \mathbb{E}')} \text{ (Exec Local)}$$

This implies that any possible local evaluation can be a global transition.

2.2.4 Formulation of Evaluation of Go Expressions

When a go expression is evaluated, the innermost undock expression surrounding the go expression is moved to the destination and the rest of the expression without it still remains at the departure. This semantics requires that:

1. division of the whole expression into the migrating part and the remaining part, and

2. creation of a new thread for the migrating part.

Suppose that expression of the form $E[(\mathtt{go}\ q)]$ is being evaluated at location p, and let $\langle A[(\mathtt{go}\ q)]\rangle$ be the innermost undock expression surrounding the $(\mathtt{go}\ q)$ such that $E[(\mathtt{go}\ q)] \equiv R[\langle A[(\mathtt{go}\ q)]\rangle]$ for some context R. Then $A[(\mathtt{go}\ q)]$ will move to the destination q and then be evaluated there. We define that a go expression is reduced to *the departure location* at the destination. In sum, the evaluation rule for a go expression at the departure is defined as follows:

$$R[\langle A[\mathtt{go}\ q]\rangle] \longrightarrow A[p].$$

where R is an arbitrary evaluation context and A is an arbitrary undock-free evaluation context.

On the other hand, the rest of the expression R stays at the departure. We define that a go expression is reduced to *the destination location* at the departure. Namely, we have that:

$$R[\langle A[\mathtt{go}\ q]\rangle] \longrightarrow R[q].$$

Now we define the semantics of go expressions as follows:

$$\frac{\begin{array}{c}(p, R[\langle A[\mathtt{go}\ q]\rangle]) \in \mathbb{E} \quad p \neq q \quad H = \mathbb{H}(p) \quad H' = \mathbb{H}(q) \\ \emptyset, \emptyset, \emptyset, A[p] \Rightarrow_{H,p,q} \sigma, \Delta, H_d, A'[p] \quad \mathbb{H}' = \mathbb{H}.\{p \mapsto \sigma H, q \mapsto H' \uplus H_d\} \\ \mathbb{E}' = \mathbb{E} \setminus \{(p, R[\langle A[\mathtt{go}\ q]\rangle])\} \cup \{(p, R[q]), (q, A'[p])\}\end{array}}{(\mathbb{H}, \mathbb{E}) \rightarrow (\mathbb{H}', \mathbb{E}')} \text{ (Exec Go 1)}$$

$$\frac{\begin{array}{c}(p, R[\langle A[\mathtt{go}\ p]\rangle]) \in \mathbb{E} \quad H = \mathbb{H}(p) \quad \emptyset, \emptyset, \emptyset, A[p] \Rightarrow_{H,p,p} \sigma, \Delta, H_d, A'[p] \\ \mathbb{H}' = \mathbb{H}.\{p \mapsto \sigma H \uplus H_d\} \\ \mathbb{E}' = \mathbb{E} \setminus \{(p, R[\langle A[\mathtt{go}\ q]\rangle])\} \cup \{(p, R[p]), (p, A'[p])\}\end{array}}{(\mathbb{H}, \mathbb{E}) \rightarrow (\mathbb{H}', \mathbb{E}')} \text{ (Exec Go 2)}$$

where A is any undock-free evaluation context and R is any evaluation context.

2.3 Expressive Power

In this section, we demonstrate that several language constructs for distributed mobile computation can be encoded in our calculus and that it can emulate distinctive features of several mobile languages.

2.3.1 Remote Evaluation

As shown, asynchronous remote evaluation can be encoded in a simple form as follows:

$$(\mathsf{remote\text{-}run!}\ p\ e) \equiv \langle \mathsf{let}\ d = (\mathsf{go}\ p)\ \mathsf{in}\ e \rangle$$

for some d is not free in e. The expression in the left side migrates to p, and then e is evaluated in the destination. Since the let-expression is enclosed within an undock expression, expressions containing the remote-run! do not migrate.

In case of synchronous remote evaluation, the mechanism of delivering the result value must be provided. We could add such a mechanisms like communication channel to our calculus as a primitive operation, but we encode them. The following syntactic macros are used to encode primitive operations for a simple communication channel:

$$\begin{aligned}
\mathsf{make_channel} \equiv \lambda x.\, \mathsf{let}\ \ a &= \mathsf{new}(\mathbf{resident})0,\\
f &= \lambda x.!a(),\\
d &= \mathsf{set}(?a)af\\
\mathsf{in}\ a&\\
\mathsf{put\text{-}value} \equiv \lambda cv.\mathsf{set}(?c)c(\lambda x.v)&\\
\mathsf{get\text{-}value} \equiv \lambda c.!c()&
\end{aligned}$$

The macro make_channel creates new address a of resident protocol, then stores function f onto the address. Once some value is applied to the function, it applies the unit value to the content of address a, that is f itself. This repetition repeats until some value is stored to the address. The macro put-value stores a value to a given address. The macro get-value invokes the channel, and waits until some value is stored at the address.

A macro that evaluates an expression at a remote location is defined as follows:

$$\begin{aligned}
\mathsf{reval}(p, e) \equiv\ \mathsf{let}\ \ c &= \mathsf{make_channel}()\\
d &= \langle\, \mathsf{let}\ r = (\mathsf{go}\ p), v = e\\
&\qquad \mathsf{in}\ \langle \mathsf{let}\ d = (\mathsf{go}\ r)\ \mathsf{in}\ \mathsf{put\text{-}value}cv \rangle \rangle\\
\mathsf{in}\ \mathsf{get\text{-}value}c&
\end{aligned}$$

This macro defines evaluation of expression e at location p. To evaluate e at location p, a channel is created and the execution at the current location waits until a return value will be stored in the channel. The inside of the first undock expression goes to location p, evaluates e there, returns to its birthplace and finally stores a return value into the channel. The control surely returns to its birthplace because the go expression $(\mathsf{go}\ p)$ is reduced to its departure in the destination. In the variable r, its departure location is stored.

2.3.2 Remote Procedure Call

The encoding of remote procedure call is simpler than that of remote evaluation because remote procedure call (RPC) does not require code to be moved at the time of invocation, but a service of RPC is assumed to be provided at the destination host. The mechanism that associates a service with its actual implementation is

called a naming service. When a remote procedure is invoked by means of RPC, the procedure is designated by its name registered in the naming server. We represent the name of a public service by a ubiquitous address. The actual implementation of a service u is assumed to be stored at address u in the destination. The semantics of RPC is thus given as follows:

$$\mathrm{RPC}(p, u, v) \equiv \mathsf{reval}(p, !uv).$$

where v is the parameter for the service. Because the implementation of the service (it is a function) is stored at address u, expression $!uv$ invokes the service with argument v.

The communication mechanism for distributed hosts in Facile [61, 62], that is an extension of Standard ML for distributed computation, is based on RPC although the mechanism can transmit a functional closure as an argument for RPC in the same way as basic values.

2.3.3 Encoding Obliq RMI

Here we describe the semantics of remote reference and remote method invocation in Obliq [16]. Obliq [16] is a simple distributed object-oriented language, which has a significant feature that one can specify the locations of values and controls in a completely implicit manner. The distinction between local and remote references is thus implicit. Even when the programmer does not care the actual location of value, the semantics of a program does not change. In other words, the semantics of Obliq is defined to be location transparent except some location-dependent constructs.

The values of Obliq is divided into immutable values such as integers and functions, and mutable ones such as objects and reference cells. When an immutable value is transmitted to a remote host, it is duplicated. When a mutable value is transmitted to a remote host, it is not duplicated, but a remote reference is created that points to the original value. The consistency of a mutable value is guaranteed since it is never duplicated. In our calculus, an immutable value can be considered as a value of copy protocol, and a mutable value can be considered as a value of resident protocol.

Mobility of a value is defined according to its mutability. The difference of execution site is specified by using the difference between a function and a method. In Obliq, a function is always executed locally since a function is copied across sites as a function is an immutable value. On the other hand a method is executed remotely since an object and of course its methods are referred as remote references from the outside of its resident site.

The location transparency of Obliq can be paraphrased in our calculus by defining semantics of remote references as follows:

(dereference of a remote reference)	$!a@p \stackrel{\mathrm{def}}{=} \mathsf{reval}(p, !a)$
(update of a remote reference)	$\mathsf{set}(P)a@pv \stackrel{\mathrm{def}}{=} \mathsf{reval}(p, \mathsf{set}(P)!av)$
(location of a remote reference)	$\mathsf{loc}(a@p) \stackrel{\mathrm{def}}{=} p$
(remote function invocation)	$\mathsf{funcall}(a@p, v) \stackrel{\mathrm{def}}{=} \mathsf{reval}(p, !a)v$
(remote method invocation)	$\mathsf{methcall}(a@p, v) \stackrel{\mathrm{def}}{=} \mathsf{reval}(p, !av)$

The semantics of dereference and update of remote references is a straightforward extension of the semantics for local references. The difference between function and

method invocations is shown by the difference between $\mathsf{reval}(p, !a)v$ and $\mathsf{reval}(p, !av)$. In the former case, the argument v is not transmitted to the destination, but the body of a function is fetched to the current location. The invocation of the function is taken place at the current location. In the latter case, the argument is sent to the destination. The invocation of the method is taken place at the destination. The operator loc retrieves the actual location of the content of a remote reference. It is naturally extended to the case of local references such that $\mathsf{loc}(a)$ is evaluated to the current location where the expression is evaluated.

2.3.4 Encoding Java Class Loader

In Java, a program can start its execution even if all the constituent class files are not loaded into the runtime system [65]. An unloaded class file will be loaded on demand. When an unloaded class file is on a remote host, it causes transmission of the class file via network. We encode this class loading mechanism of Java by an example of a function that is not copied from a remote host until it is invoked. To describe such a code movement mechanism, let us consider the following heaps:

$$\begin{array}{l}\text{server site } p : \{a \mapsto (\mathbf{resident}, f)\} \\ \text{client site } q : \{s \mapsto (\mathbf{copy}, \lambda x.\mathsf{let}\ d = \mathsf{set}(\mathbf{resident})s(!(a@p))\ \mathsf{in}\ !sx)\}\end{array}$$

In this diagram, the actual implementation of the unloaded function is f. It is in the server heap yet. In the client heap, there is a stub s for f instead of f. When the stub is invoked with argument v, the evaluation at location q proceeds as follows:

$$\begin{array}{ll} & \mathsf{letrec}\ H\ \mathsf{in}\ !sv \\ \rightsquigarrow & \mathsf{letrec}\ H\ \mathsf{in}\ (\lambda x.\mathsf{let}\ d = \mathsf{set}(\mathbf{resident})s(!(a@p))\ \mathsf{in}\ !sx)v \\ \rightsquigarrow & \mathsf{letrec}\ H\ \mathsf{in}\ \mathsf{let}\ d = \mathsf{set}(\mathbf{resident})s(!(a@p))\ \mathsf{in}\ !sv \\ \rightsquigarrow & \mathsf{letrec}\ H\ \mathsf{in}\ \mathsf{let}\ d = \mathsf{set}(\mathbf{resident})sf\ \mathsf{in}\ !sv \\ \rightsquigarrow & \mathsf{letrec}\ \{s \mapsto (\mathbf{resident}, f)\}\ \mathsf{in}\ \mathsf{let}\ d = ()\ \mathsf{in}\ !sv \\ \rightsquigarrow & \mathsf{letrec}\ \{s \mapsto (\mathbf{resident}, f)\}\ \mathsf{in}\ !sv \\ \rightsquigarrow & \mathsf{letrec}\ \{s \mapsto (\mathbf{resident}, f)\}\ \mathsf{in}\ fv \end{array}$$

This derivation is somewhat incorrect since the evaluation of $!(a@p)$ requires remote evaluation and thus global transition, but this abuse is just for simplicity. When the stub is invoked with a value v as $!sv$, it copies the function f to the client heap and then invoke f with v.

A stub of a function f can be generated by evaluating the following macro at the server site:

$$\begin{array}{lll}\mathsf{genstub}(f) \equiv & \mathsf{let} & a = \mathsf{new}(\mathbf{resident})f, \\ & & s = \mathsf{new}(\mathbf{copy})0, \\ & & f' = \lambda x.\mathsf{let}\ d = \mathsf{set}(\mathbf{resident})s(!a)\ \mathsf{in}\ !sx, \\ & & d = \mathsf{set}(\mathbf{copy})sf' \\ & \mathsf{in}\ s & \end{array}$$

When $\mathsf{genstub}(f)$ is evaluated at the server site, a stub of f is generated. When a stub is transmitted to a client site and then is stored into a heap, these heaps look like the ones as shown above.

2.3.5 Encoding Kali-Scheme Proxy Values

Kali-Scheme [21] is a distributed implementation of Scheme interpreter that allows transmission of higher-order values including continuations. When values are transmitted to a remote host, they are fully duplicated provided that sharing within the transmitted values is preserved. Kali-scheme provides a proxy value, which is a first-class object in Kali-Scheme used to share values among different locations. A proxy value has its own content as a cons cell has the car value and the cdr value. In addition, a proxy value remembers the place where it was created. If a proxy is transmitted to another location, the value (content) is just duplicated there. Eventually, when it comes back to its birth location, the proxy value is not copied from the immediate departure, but the reference to the proxy value is connected to the original proxy value. The programmer can update the content of a proxy value by specifying its evaluation location explicitly like:

$$(\mathsf{remote\text{-}run!}\ (\mathsf{proxy\text{-}creator}\ \text{proxy})\ \mathsf{set\text{-}proxy\text{-}value!}\ \text{proxy new-value})$$

where proxy-creator returns the location where the proxy is created and set-proxy-value! updates the content of the proxy value.

The behavior of a proxy value can be simulated in our calculus. The constructs manipulating proxy values are given as follows:

$$\begin{aligned}
(\mathsf{make\text{-}proxy}\ x) &= \mathsf{let}\ \text{proxy} = \mathsf{new}(\mathbf{resident})0 \\
&\quad\ \mathsf{in}\ (\mathsf{assoc\text{-}set}\ \mathsf{proxy\text{-}table}\ \text{proxy}\ x; \text{proxy}) \\
(\mathsf{proxy\text{-}creator}\ \text{proxy}) &= (\mathsf{loc}\ \text{proxy}) \\
(\mathsf{proxy\text{-}value}\ \text{proxy}) &= (\mathsf{assoc\text{-}ref}\ \mathsf{proxy\text{-}table}\ \text{proxy}) \\
(\mathsf{set\text{-}proxy\text{-}value}\ \text{proxy}\ v) &= (\mathsf{assoc\text{-}set}\ \mathsf{proxy\text{-}table}\ \text{proxy}\ v)
\end{aligned}$$

where make-proxy is an operation for creating a new proxy and proxy-value retrieves the value of a proxy. Association lists are used to maintain the mapping from a proxy value to its content. The association list is supposed to be provided for each location and is referred to by ubiquitous reference proxy-table. The operation assoc-ref retrieves the content of the association list designated by the second argument. The operation assoc-set updates the content of the association list.

2.3.6 Encoding Telescript Ownership Mechanism

Telescript has a notion of *ownership* [41], which is used to specify the area of values to be transmitted on migration and is used for garbage collection. There are two kinds of reference in Telescript: *owner* reference and *non-owner* reference. Intuitively, a value that is referred to by an owner reference is considered to be possessed by its referrer. When an object (agent in terms of Telescript) migrates to another location, all the values owned by the object will be also taken there and all the references pointing to non-owned values in the departure location are disconnected or nullified. The ownership mechanism based on owner and non-owner references can be described in our calculus by using data movement protocols. An owner reference is a reference of takeaway protocol and a non-owner reference is a reference of proper protocol.

2.4 Controllable Migration Transparency

In this section, we develop the *control aspect* of our mobile calculus especially for *implementing* migration. When a program execution is migrated, the continuation of the program execution is transmitted to the destination. We investigate how the continuation of a program execution is identified and transmitted on a single migration, which is quite important for implementing migration. This is the reason why we develop a *sequential* execution model in this section although mobile computation usually accommodates concurrency.

From the same reason, we focus on migration of code and execution states in this section. Migration from the viewpoint of data is examined in Section 2.5. A migration consists of the following steps of procedure:

1. Freeze the current thread (that has just brought about the migration).
2. Identify the area of code and execution state to be transmitted.
3. Transmit the necessary stuffs to the destination of the migration.
4. If succeed, restart the thread at the departure and invoke a new thread for the migrated program execution at the destination in parallel.

To fulfill the semantics of migration in our calculus, this procedure of migration must be atomic: these steps of the migration procedure must not be intervened or interfered with another migration.

2.4.1 *Partial* Continuation

Continuation [91] is an important notion for understanding control in programs. Although we have often said like "the continuation of a program execution is transmitted on migration", exactly speaking, this continuation in our mobile calculus is not the usual, ordinary continuation, but a *partial* continuation [29, 34, 35, 56]. An ordinary continuation represents the *entire* rest of computation: it is a mapping from a value to the final answer, it cannot be invoked more than once, it cannot be composed and the control never returns to the point of invocation unless it is captured by another continuation. But a partial continuation is a functional prefix of the ordinary continuation: it can be invoked more than once as the ordinary function, it can be composed and the evaluation of a partial continuation will return to the point of invocation unless another invocation of continuation intervenes. In case of program migration, the ordinary continuation do not be moved. What is migrated is usually a part of the entire program. Remember that the portion to be migrated is denoted by an undock expression $\langle e \rangle$ in our mobile calculus. The notion of partial continuation is closely related to code migration.

There are two major slightly different notions of partial continuation, which are Felleisen's [34, 35] and Danvy's [29]. Felleisen's notion of partial continuation has the dynamic flavor. It precisely corresponds to the execution stack. In contrast, Danvy's notion has the static flavor. It has an (extended) continuation semantics. Their notions are much alike each other. They coincide in most cases. But there are some examples that reveals their difference [28]. We describe our code migration by using partial continuation for both cases.

2.4.2 Felleisen's Notion of Partial Continuation

We overview Felleisen's (modified) λ_v-C-calculus [34] briefly. The term language and the evaluation contexts are defined by the following grammar:

$$\begin{aligned} v &::= c \mid x \mid (\lambda x.e), \\ e &::= v \mid (ee) \mid (\mathcal{F}e) \mid (\#e), \\ E &::= [\,] \mid (vE) \mid (Ee) \mid (\#E), \\ A &::= [\,] \mid (vA) \mid (Ae). \end{aligned}$$

The language is an extension of the standard λ-calculus with two operators $\mathcal{F}$ and $\#$. The value v consists of constant c, variable x and lambda expression $\lambda x.e$. The expression e consists of value v, application (ee), $\mathcal{F}$ application $(\mathcal{F}e)$ and $\#$ (called *prompt*) application $(\#e)$. The evaluation context denoted by A is called a *prompt-free* evaluation context.

Intuitively, the $\mathcal{F}$ operator captures the current continuation and applies it to the argument of the $\mathcal{F}$. The call/cc operator can be encoded by using $\mathcal{F}$ as follows:

$$(\text{call/cc } e) = (\mathcal{F}\lambda k.k(ek)).$$

The role of $\#$ operator is to enclose the effect of $\mathcal{F}$ within $\#$. For instance, $(\mathcal{F}\lambda k.1)$ is the term that aborts the current continuation and evaluates abruptly to integer 1. The current continuation of this term is captured and is applied to $\lambda k.1$. The current continuation is bound to variable k and is just ignored. But If this term is enclosed by $\#$, the reduction proceeds differently:

$$E[\mathcal{F}\lambda k.1] \to 1,$$

$$E'[\#E[\mathcal{F}\lambda k.1]] \to E'[1].$$

where E is any prompt-free evaluation context and E' is any evaluation context.

The reduction relations are the following six ones:

$$\begin{aligned} cc' &\to_F \delta(c, c'), && (\delta_F) \\ (\lambda x.e)v &\to_F e\{x \leftarrow v\}, && (\beta_F) \\ (\mathcal{F}e)e' &\to_F \mathcal{F}(\lambda k.e(\lambda m.k(me'))), && (\mathcal{F}_L) \\ v(\mathcal{F}e) &\to_F \mathcal{F}(\lambda k.e(\lambda m.k(vm))), && (\mathcal{F}_R) \\ (\#(\mathcal{F}e)) &\to_F (\#(e(\lambda x.x))), && (\#_{\mathcal{F}}) \\ P(\#v) &\to_F v. && (\#_v) \end{aligned}$$

This system is called λ_v-C-calculus. It has the Church-Rosser property. The standard reduction relation defines evaluation step $e \mapsto_F e'$ if there are terms f and f' and evaluation context E such that $e \equiv E[f]$, $e' \equiv E[f']$ and $f \to_F f'$. Its reflexive and transitive closure is denoted by $\mapsto_F^*$. If e has no further reduction sequences, we say that e terminates.

Definition 2.4.1. Let e be a term in λ_v-C-calculus. We say that e *terminates*, denoted $e\downarrow$, if there is no e' such that $e \mapsto_F e'$.

Two terms e and e' are *operationally equivalent* if they are indistinguishable when one is replaced with the other in any evaluation contexts.

Definition 2.4.2. Let e and e' be terms in λ_v-C-calculus. We say that e and e' are *operationally equivalent*, denoted by $e \simeq e'$, if (1) whenever $E[e]\downarrow$, then we also have $E[e']\downarrow$ and vice versa, and (2) whenever $E[e] \mapsto_F^* c$ for some constant c, then we also have $E[e'] \mapsto_F^* c$, and vice versa, for any evaluation context E such that $E[e]$ and $E[e']$ are closed.

Definition 2.4.3. Let e be a term. $\mathrm{eval}_F(e) = v$ iff $e \mapsto_F^* v$.

2.4.3 Danvy's Notion of Partial Continuation

Next, we overview Danvy's calculus developed in Ref. [29]. The term language is defined by the following grammar:

$$e ::= x \mid \lambda x.e \mid (ee) \mid (\xi x.e) \mid \langle e \rangle.$$

The expression e consists of variable x, lambda expression $\lambda x.e$, application (ee), shift expression $\xi x.e$ and reset expression $\langle e \rangle$. The shift expression $(\xi x.e)$ nearly corresponds to Felleisen's $\mathcal{F}$. By evaluating a shift expression $(\xi x.e)$, the current continuation is captured and the body e is evaluation in the environment where x is bound to the continuation. The reset expression $\langle e \rangle$ nearly corresponds to Felleisen's $\#$. Both delimit the effect of the control operators.

The semantics of the expression is given in the form of a so-called meta-continuation semantics:

$$\begin{aligned}
\mathcal{E}[\![x]\!]\rho\kappa\gamma &= \kappa(\rho(x))\gamma \\
\mathcal{E}[\![e_1 e_2]\!]\rho\kappa\gamma &= \mathcal{E}[\![e_1]\!]\rho(\lambda f\gamma'.\mathcal{E}[\![e_2]\!]\rho(\lambda a\gamma''.fa\kappa\gamma'')\gamma')\gamma \\
\mathcal{E}[\![\lambda x.e]\!]\rho\kappa\gamma &= \kappa(\lambda v\kappa'\gamma'.\mathcal{E}[\![e]\!]\rho.\{x \mapsto v\}\kappa'\gamma')\gamma \\
\mathcal{E}[\![\xi x.e]\!]\rho\kappa\gamma &= \mathcal{E}[\![e]\!]\rho.\{x \mapsto \lambda v\kappa'\gamma'.\kappa v(\lambda w.\kappa' w\gamma')\}(\lambda x\gamma''.\gamma'' x)\gamma \\
\mathcal{E}[\![\langle e \rangle]\!]\rho\kappa\gamma &= \mathcal{E}[\![e]\!]\rho(\lambda x\gamma'.\gamma' x)(\lambda v.\kappa v\gamma)
\end{aligned}$$

The semantics of term e is given in the form $\mathcal{E}[\![e]\!]\rho\kappa\gamma$ where ρ is an environment, κ is a continuation and γ is a meta continuation. According to Ref. [29], the relation between this meta-continuation semantics and an ordinary continuation semantics is presented by the following equation:

$$\mathcal{E}[\![e]\!]\rho\kappa\gamma = \gamma(\mathcal{E}_c[\![e]\!]\rho_c\kappa).$$

where $\mathcal{E}_c$ defines the semantic function of the ordinary continuation semantics and ρ_c is the environment correspondent to ρ in the ordinary continuation semantics. Intuitively, the extra term γ in the meta-continuation semantics represents the effect *after* the final value of expression e is obtained. The final value, that is equal to $\mathcal{E}_c[\![e]\!]\rho_c\kappa$ in the ordinary continuation semantics, is applied to γ.

Danvy also gives an extension of continuation-passing style (CPS) translation for his constructs for partial continuation. The CPS translation [8, 60] is a translation often used in compiling a program in functional programming languages. The translation function $\lceil\ \rceil_C$ are defined as follows:

$$\begin{aligned}
\lceil x \rceil_C &= \lambda\kappa.\kappa x, \\
\lceil e_1 e_2 \rceil_C &= \lambda\kappa.\lceil e_1 \rceil_C(\lambda f.\lceil e_2 \rceil_C(\lambda a.fa\kappa)), \\
\lceil \lambda x.e \rceil_C &= \lambda\kappa.\kappa(\lambda x.\lceil e \rceil_C),
\end{aligned}$$

$$\lceil \xi x.e \rceil_C = \lambda\kappa.[(\lambda a\kappa'.\kappa'(\kappa a))/x]\lceil e \rceil_C(\lambda y.y),$$
$$\lceil \langle e \rangle \rceil_C = \lambda\kappa.\kappa(\lceil e \rceil_C(\lambda x.x))$$

provided that new bound variables introduced by the translation are assumed not to collide with existing ones. This translation translates a term in Danvy's language for partial continuation to the one in a standard lambda calculus.

2.4.4 A Sequential Subset of our Mobile Calculus

To see the close relationship between the operators for migration (go and $\langle\rangle$) and those for partial continuation, a sequential subset of our mobile calculus is extracted. The point we want to investigate here is the control in a sequential part of our mobile calculus. The term of the sequential calculus and the evaluation contexts are defined by the following grammar:

$$\begin{aligned} v &::= \mathsf{u} \mid p \mid x, \\ e &::= v \mid \lambda x.e \mid (ee) \mid (\mathsf{go}\ e) \mid \langle e \rangle \mid @e, \\ E &::= [\,] \mid (\lambda x.e)E \mid Ee \mid (\mathsf{go}\ E) \mid \langle E \rangle \mid @E, \\ A &::= [\,] \mid (\lambda x.e)A \mid Ae \mid (\mathsf{go}\ A) \mid @A. \end{aligned}$$

The value v includes the special constant u, called the *unit*, location p and variable x. The expression e includes value v, lambda expression $(\lambda x.e)$, application (ee), go expression $(\mathsf{go}\ e)$, undock expression $\langle e \rangle$ and location expression $(@e)$. A general evaluation context is denoted by E while an undock-free evaluation context is denoted by A.

Intuitively, location expression of the form $(@pe)$ indicates that the evaluation of expression e proceeds in location p. If an expression is in the scope of multiple, overlapped location expressions, only the innermost location is valid.

The reduction relations are defined as follows:

$$\begin{aligned} (\lambda x.e)v &\to_M [v/x]e \\ \langle A[\mathsf{go}\ p] \rangle &\to_M @p(A[\mathsf{u}]) \\ \langle v \rangle &\to_M v \\ @pv &\to_M v \end{aligned}$$

The evaluation of a go expression causes the program execution to be migrated. The term $\langle A[\mathsf{go}\ p] \rangle$ is thus translated into $@p(A[\mathsf{u}])$ since it represents that the expression $A[\mathsf{u}]$ is evaluated in location p. The standard reduction relation $e \mapsto_M e'$ is defined if there are terms f and f' and evaluation context E such that $e \equiv E[f]$, $e' \equiv E[f']$ and $f \to_M f'$. Its reflexive and transitive closure is denoted by $\mapsto_M^*$. The operational equivalence is defined in the completely analogous way as Felleisen's case. Evaluation function $\mathrm{eval}_M(e)$ is defined to be v if and only if $e \mapsto_M^* v$. This system is called Λ_M.

The relationship between the operators for migration and those for partial continuation is presented in quite a direct form. A simple translation function is given for each calculus for partial continuation, that is a mapping from a term in the above sequential calculus to a term in the calculus for partial continuation.

2.4.5 Translation to Felleisen's System

A translation function from a term in the sequential mobile calculus to the one in Felleisen's calculus is presented. The translation function $\lceil\ \rceil_F$ is defined as follows:

$$\begin{aligned}
\lceil[]\rceil_F &= [],\\
\lceil \mathsf{u} \rceil_F &= \mathsf{u},\\
\lceil p \rceil_F &= p,\\
\lceil x \rceil_F &= x,\\
\lceil \lambda x.e \rceil_F &= \lambda x.\lceil e \rceil_F,\\
\lceil e_1 e_2 \rceil_F &= \lceil e_1 \rceil_F \lceil e_2 \rceil_F,\\
\lceil \mathsf{go}\ e \rceil_F &= (\lambda x.\mathcal{F}(\lambda a.(@x(a\mathsf{u}))))\lceil e \rceil_F,\\
\lceil \langle e \rangle \rceil_F &= \#\lceil e \rceil_F,\\
\lceil @e \rceil_F &= \#@\lceil e \rceil_F
\end{aligned}$$

provided that new bound variables introduced by the translation are assumed not to collide with existing ones. The meaning of the constant symbol @ in Felleisen's calculus is given by the function δ as follows:

$$\delta(@, p) = \lambda x.x$$

and δ is undefined otherwise. The relationship between the operators for migration and those for partial continuation is embodied in the translation rules for go and undock expressions. In the translation rules for go expressions, a go expression is translated into the term like:

$$(\lambda x.\mathcal{F}(\lambda a.(@x(a\mathsf{u}))))\lceil e \rceil_F.$$

The argument e for the go expression is the destination of migration. It is applied to the first half of the term and then is bound to variable x. Then $\mathcal{F}$ operator captures the current partial continuation up to the innermost prompt expression and applies it to the subterm $(\lambda a.(@x(a\mathsf{u})))$. The current partial continuation is bound to variable a and the term left behind is $@x(a\mathsf{u})$. The current partial continuation a is invoked at the location designated by variable x. This interpretation of migration agrees with our intuition for migration. An undock expression is translated directly to a prompt expression: an undock expression is exactly the operator that delimits the effect of control, the effect of migration. We call this flexible form of code migration *controllable transparent migration.*

Several formal properties of the translation function are presented to show the validity of the translation function.

Lemma 2.4.1. *If E is an evaluation context in Λ_M, then $\lceil E \rceil_F$ is also an evaluation context in λ_v-C-calculus such that $\lceil E[e] \rceil_F =_\alpha \lceil E \rceil_F [\lceil e \rceil_F]$ for any term e. Moreover, if E is an undock-free evaluation context, then $\lceil E \rceil_F$ is also a prompt-free evaluation context.*

Proof. The proof is by induction on the structure of E. □

Lemma 2.4.2. *Let A be a prompt-free evaluation context in λ_v-C-calculus. Then we have that $A[\mathcal{F}e] \mapsto_F^* \mathcal{F}e'$ where e' is a term such that $e'(\lambda x.x) \to_{\alpha\beta}^* e(\lambda y.A[y])$.*

Proof. The proof is by induction on the structure of A.
Case ($A \equiv [\,]$) Obviously, $\mathcal{F}e \mapsto_F^* \mathcal{F}e$ and $e(\lambda x.x) \equiv e(\lambda x.x)$.
Case ($A \equiv A_0[v_0[\,]]$) We have that $A_0[v_0(\mathcal{F}e)] \mapsto_F A_0[\mathcal{F}(\lambda k.e(\lambda m.k(v_0 m)))]$ by $(\mathcal{F}_R)$. By induction hypothesis, we have that $A_0[\mathcal{F}(\lambda k.e(\lambda m.k(v_0 m)))] \mapsto_F \mathcal{F}e'$ such that $e'(\lambda x.x) \to_{\alpha\beta}^* (\lambda k.e(\lambda m.k(v_0 m)))(\lambda y.A_0[y])$. By (β_F), we have that $(\lambda k.e(\lambda m.k(v_0 m)))(\lambda y.A_0[y]) \mapsto_F e(\lambda m.A_0[v_0 m])$. Obviously, $\lambda x.A[x] =_\alpha \lambda m.A_0[v_0 m]$.
Case ($A \equiv A_0[[\,]e_0]$) The proof is analogous to the previous case. □

Lemma 2.4.3. *Let e be a term in Λ_M. If $e \mapsto_M e'$, then $\lceil e \rceil_F \mapsto_F^* \lceil e' \rceil_F$.*

Proof. The proof is by case analysis on the reduction relation $e \mapsto_M e'$. We show only the case for a go expression. By hypothesis, there are an evaluation context E and an undock-free evaluation context A such that $e \equiv E[\langle A[\mathsf{go}\ p]\rangle]$ and $e' \equiv E[@p(A[\mathsf{u}])]$. We have that:

$$\begin{aligned}
\lceil \langle A[\mathsf{go}\ p]\rangle \rceil_F &= \lceil \langle A\rangle \rceil_F[\lceil \mathsf{go}\ p\rceil] \qquad \text{(by Lemma 2.4.1)}\\
&= \#\lceil A\rceil_F[(\lambda x.\mathcal{F}(\lambda a.@x(a\mathsf{u})))p]\\
&\to_F \#\lceil A\rceil_F[\mathcal{F}(\lambda a.@p(a\mathsf{u}))] \qquad (\text{by } \beta_F)\\
&\to_F^* \#(\mathcal{F}e') \qquad \text{(by Lemma 2.4.2)}\\
&\to_F \#(e'(\lambda x.x)) \qquad (\text{by } \#_{\mathcal{F}})\\
&\to_{\alpha\beta}^* \#((\lambda a.@p(a\mathsf{u}))(\lambda y.\lceil A\rceil_F[y])\\
&\to_F \#(@p(\lceil A\rceil_F[\mathsf{u}])) \qquad (\text{by } \beta_F).
\end{aligned}$$

On the other hand, $\lceil @p(A[\mathsf{u}])\rceil_F = \#(@p(\lceil A\rceil_F[\mathsf{u}]))$. By Lemma 2.4.1, $\lceil E\rceil_F$ is also an evaluation context in λ_v-C-calculus. Therefore, we have that $\lceil E\rceil_F[\lceil\langle A[\mathsf{go}\ p]\rangle\rceil_F] \to_F^* \lceil E\rceil_F[\lceil @p(A[\mathsf{u}])\rceil_F]$. □

Lemma 2.4.4. *Let e be a term in Λ_M. If $e\!\downarrow$, then $\lceil e\rceil_F\!\downarrow$.*

Proof. To satisfy $e\!\downarrow$, we must have either:

1. there is an evaluation context E such that $e \equiv E[xv]$.
2. there is an evaluation context E such that $e \equiv E[pv]$ or $e \equiv E[\mathsf{u}v]$.
3. there is an evaluation context E such that $e \equiv E[@v]$ and v is not a location.

In the first case, $\lceil E\rceil_F[x\lceil v\rceil_F]$ also gets stuck. In the second case, $\lceil E\rceil_F[p\lceil v\rceil_F]$ and $\lceil E\rceil_F[\mathsf{u}\lceil v\rceil_F]$ also get stuck. In the third case, e also gets stuck because $\delta(@, v)$ is undefined. □

The translation function $\lceil\ \rceil_F$ is correct in the sense of the following two propositions.

Proposition 2.4.1 (Simulation). *Let e be a term in Λ_M. $\lceil eval_M(e)\rceil_F = eval_F(\lceil e\rceil_F)$.*

Proof. From Lemma 2.4.3, whenever $e \mapsto_M^* e'$, we also have that $\lceil e\rceil_F \mapsto_F^* \lceil e'\rceil_F$. From Lemma 2.4.3 and Lemma 2.4.4, $e\!\downarrow$ if and only if $\lceil e\rceil_F\!\downarrow$. □

A translation is adequate if the equivalence of the translated terms implies the equivalence of the original terms. The reason why the inverse does not hold is that $\mathcal{F}$ operator can appear anywhere in an evaluation context in λ_v-C-calculus. Partial continuation in the mobile calculus is used only in quite a restricted form.

Proposition 2.4.2 (Adequacy). *Let e and e' terms in Λ_M. $e \simeq e'$ if $\lceil e \rceil_F \simeq \lceil e' \rceil_F$. The inverse does not hold.*

Proof. Suppose that $\lceil e \rceil_F \simeq \lceil e' \rceil_F$ and $e \not\simeq e'$. Then there must exist an evaluation context E satisfying either one of the following cases:

1. $E[e]\downarrow$ and $E[e']$ does not terminate.
2. $\mathrm{eval}_M(E[e]) = c$ for some constant c and $\mathrm{eval}_M(E[e']) = v'$ for some value v' such that $c \neq v'$.

1. By Proposition 2.4.1, we have that $E[e]\downarrow$ if and only if $\lceil E[e] \rceil_F \downarrow$ and that $E[e']\downarrow$ if and only if $\lceil E[e'] \rceil_F \downarrow$. By hypothesis, $\lceil E[e] \rceil_F \downarrow$ if and only if $\lceil E[e'] \rceil_F \downarrow$. Thus we must have that $E[e]\downarrow$ if and only if $E[e']\downarrow$. 2. If $c \neq v'$, then $\lceil c \rceil_F \neq \lceil v' \rceil_F$. Thus we cannot have $\lceil e \rceil_F \simeq \lceil e' \rceil_F$.

A counterexample for the inverse of the proposition is that $e \equiv \lambda x.(\langle 1 + (x0) \rangle 0)$, $e' \equiv \lambda x.(\langle 2 + (x0) \rangle 0)$ and $E \equiv [](\lambda y.\mathcal{F}(\lambda kz.k0))$. Here we assume that the addition $(+)$ is defined appropriately. Then, we have $e \simeq e'$, but $\mathrm{eval}_F(E[\lceil e \rceil_F]) = 1$ and $\mathrm{eval}_F(E[\lceil e' \rceil_F]) = 2$. □

2.4.6 Translation to Danvy's System

A translation function from a term in the sequential mobile calculus to the one in Danvy's calculus is presented. The translation function $\lceil\ \rceil_D$ is defined as follows:

$$
\begin{aligned}
\lceil [] \rceil_D &= [], \\
\lceil \mathsf{u} \rceil_D &= \mathsf{u}, \\
\lceil p \rceil_D &= p, \\
\lceil x \rceil_D &= x, \\
\lceil \lambda x.e \rceil_D &= \lambda x.\lceil e \rceil_D, \\
\lceil e_1 e_2 \rceil_D &= \lceil e_1 \rceil_D \lceil e_2 \rceil_D, \\
\lceil \mathsf{go}\ e \rceil_D &= (\lambda x.\xi a.(@x(a\mathsf{u})))\lceil e \rceil_D, \\
\lceil \langle e \rangle \rceil_D &= \langle \lceil e \rceil_D \rangle, \\
\lceil @e \rceil_D &= \langle @\lceil e \rceil_D \rangle
\end{aligned}
$$

The translation function is similar to the one from Felleisen's calculus except the notational difference. Though their calculi are essentially different with each other, the difference does not appear in describing the mobile operators. The semantics of constant @ in Danvy's calculus is defined as follows:

$$\mathcal{E}[\![@e]\!]\rho\kappa\gamma = \mathcal{E}[\![e]\!]\rho(\lambda v\gamma'.\kappa(\lambda w\kappa'\gamma''.\kappa' w\gamma'')\gamma')\gamma.$$

We state the correctness of this translation in the form of a proposition. A preliminary lemmas is needed to show the correctness.

Lemma 2.4.5. *Let E be an evaluation context in Λ_M. Then $\lceil E \rceil_D$ is also an evaluation context in Danvy's calculus such as $\lceil E[e] \rceil_D = \lceil E \rceil_D[\lceil e \rceil_D]$ for any expression e.*

Proof. The proof is by induction on the structure of e. □

Proposition 2.4.3 (Semantic Soundness). *Whenever* $e \mapsto_M e'$, *then* $\mathcal{E}[\![\lceil e \rceil_D]\!] =_{\alpha\beta\eta} \mathcal{E}[\![\lceil e' \rceil_D]\!]$.

Proof. The proof is by case analysis on the reduction $e \mapsto_M e'$. We show only the case for a go expression. By hypothesis, there are an evaluation context E and an undock-free evaluation context A such that $e \equiv E[\langle A[\mathsf{go}\ p]\rangle]$ and $e' \equiv E[@p(A[\mathsf{u}])]$. It is enough to show that $\mathcal{E}[\![\lceil\langle A[\mathsf{go}\ p]\rangle\rceil_D]\!] =_{\alpha\beta\eta} \mathcal{E}[\![\lceil @p(A[\mathsf{u}])\rceil_D]\!]$. By the definition of $\lceil\ \rceil_D$ and Lemma 2.4.5, we have that:

$$\begin{aligned}\lceil\langle A[\mathsf{go}\ p]\rangle\rceil_D &= \langle \lceil A\rceil_D[(\lambda x.\xi a.(@x(a\mathsf{u})))p]\rangle,\\ \lceil @p(A[\mathsf{u}])\rceil_D &= \langle @p(\lceil A\rceil_D[\mathsf{u}])\rangle.\end{aligned}$$

By calculation, we have that

$$\begin{aligned}\mathcal{E}[\![\lceil A\rceil_D[(\lambda x.\xi a.(@x(a\mathsf{u})))p]]\!]\rho\kappa\gamma &=_{\alpha\beta\eta} \mathcal{E}[\![\lceil A\rceil_D[x]]\!]\rho.\{x \mapsto \mathsf{u}\}(\lambda a\gamma'.\gamma' a)\gamma\\ &=_{\alpha\beta\eta} \mathcal{E}[\![@p(\lceil A\rceil_D[\mathsf{u}])]\!].\end{aligned}$$

Therefore, from Lemma 2.4.5, we have $\mathcal{E}[\![\lceil e\rceil_D]\!] =_{\alpha\beta\eta} \mathcal{E}[\![\lceil e'\rceil_D]\!]$. □

2.4.7 Application — Implementing our Mobile Calculus by Using the Extended CPS

CPS translation is useful for compiling a mobile program. Because the program transformed by CPS translation explicitly manages the continuation (the rest of computation), migration of the program execution is achieved by migrating the continuation to a remote site. A scheme using CPS translation to identify the code fragment to be migrated is used in the system called *Tube* [46]. It is an extension of Scheme that allows a program execution to be transmitted from a site to another. Because Scheme has a first-class continuation, the operator like our go expression can be defined. Knabe also applies an (ordinary) CPS translation in implementing Facile [61], but in a different sense from ours. In his system, CPS translation is used only for compiling a program in Facile and dynamic invocation of functions is not migrated.

Application of the ordinary CPS translation is, however, unsatisfactory in implementing mobile operators because of the presence of undock expressions, the operator that splits the continuation. The reason why undock expressions are introduced is twofold: the efficiency of migration and the convenience of programming. It is reported that transmission of the continuation is quite inefficient [21, 55, 61] because the continuation is nearly equal to the entire program environment: it tends to be huge data structure. Besides, when the programmer writes a mobile program, a migration of the entire program is not usually necessary. A migration of a small piece of code suffices the programmer in many cases. Therefore, some operator that delimits the effect of migration is needed (like our undock expression). Our aim is to give a scheme to enable transmission of a *part* of dynamic function invocation, using the extended CPS.

The similarity between the migration operators and partial continuation suggests the implementation schemes of migration. Here we suggest a compilation scheme by using the extended CPS translation. As presented, Danvy gives a translation from a term in his language to a pure lambda term in the form of an extension of CPS. The resulting term, of course, does not include the operators for partial continuation.

By using the translation, two-step compilation scheme can be developed. In the first step, a program in Λ_M is compiled into a term in Danvy's calculus. And in the second step, the translated term is compiled again into a term in the extended CPS. The final resulting term may include constant @ in the form $@pe$. This @ is interpreted as an issue of remote evaluation of expression e at location p.

Jagannathan proposes an operator [55] analogous to our undock expression based on Felleisen's prompt operator and the motivation to introduce such an operator agrees with our one completely. He gives an operational semantics for the language, but a compilation scheme for the language is not presented.

2.4.8 Application — Optimizing the CPS Implementation

Furthermore, this compilation scheme has possibilities for optimization. Let us consider the form of migration in which a migrated program will return to the departure location in future. In this case, migration takes place twice, thus the final compiled term includes @ twice. The second migration takes place after the first migration, which implies that the second @ is in the scope of the first @ like $@q(...@pe...)$ where p is the current (departure) location and q is the destination location. Here we can omit the transmission of e because it will be evaluation at the current location. This may reduce the cost of transmission and the cost of dynamic compilation of transmitted code. In sum, when @ appears in the scope of other @ and the destination can be inferred to be the current location, we can omit the migrated program to be transmitted.

2.5 Migration Transparency on Data

Migration is called *transparent* if the execution of a migrating program is resumed at a destination site with exactly the same execution state as that of the migration time. In existing mobile systems, a typical validation for migration transparency is based on full duplication: "our migration scheme is transparent since all the reachable values from a migrating program are duplicated onto the destination store." Because our calculus enables finer control for data migration, the migration transparency is thus unclear. In this section, we give a proof of the transparency for data migration in our calculus. Besides, we show that our data migration controlled by data movement protocol is *complete*, that is any transparent data migration can be specified by changing data movement protocols assigned to values.

These properties on data migration, however, is given for the subset of our calculus not including the proper and takeaway protocols. Because when references of these data movement protocols are transmitted on migration, the references are disconnected with their referents on an either side of migration. Obviously, the data migration is thus not transparent for these protocols.

In this section, we assume that local addresses are disjoint in different heaps. That is when H and H' are heaps in different locations, it implies that $\mathrm{Dom}(H) \cap \mathrm{Dom}(H') = \emptyset$. We also assume that every remote reference really refers to a remote value, that is the values at location p do not contain remote references $a@p$. This simplifies the following discussions without loss of generality.

2.5.1 An Infinite Tree as a Representation of a Data Structure Embedded in the Heaps

First of all, we must define the transparency of data migration, the conditions by which data migration is determined to be transparent. An equivalence relation between values in the heaps is defined for that purpose. The points that must be considered are as follows:

Recursion A data structure in the heaps can be cyclic.

Sharing A data structure in the heaps may have shared substructures.

Therefore we compare two data structures the in heaps by comparing their infinite expansions with sharing considered. The infinite expansion of a data structure is defined by using *infinite trees* [24].

A tree is a mapping from its constituent node to a pair of an alphabet of the node and a set of subnodes. An alphabet is a designator of the node. Each alphabet is assigned its arity, a non-negative integer, which defines the number of subnodes. A node is in general of the form:

$$f(t_1, ..., t_n).$$

where f is the alphabet of this node, n is the arity of f (equal to the number of its subnodes), and $t_1, ...t_n$ are its subnodes. A node of arity 0 is called a constant.

The set of alphabets used for representing a data structure in the heaps includes all expressions as constants, and the following ones:

arity 1: $\mathsf{ref}_{a@p}-, \lambda x.-, (\mathsf{go}\ -), \langle - \rangle, !-, ?-$.

arity 2: $(--), \mathsf{new}(-)-$.

arity 3: $\mathsf{set}(-)--$.

Each node of a tree may not be denoted by prefix form, but a kind of infix form $((\lambda x.-)(t)$ may be written as $\lambda x.t)$. The above alphabets of arity more than zero are corresponding to construction of the expressions. The $\mathsf{ref}_{a@p}-$ corresponds to the reference structure. Intuitively, $\mathsf{ref}_{a@p}5$ represents the data structure of a reference to address a where integer 5 is stored in the heap at location p.

2.5.2 Operations on Trees

The operations on trees necessary for the subsequent discussions are defined.

Definition 2.5.1 (The alphabets of a tree). Let t be a tree whose alphabet is f and which has n subtrees $(t_1, ..., t_n)$. For a finite sequence α of integers, the alphabet of t designated by α, written $t(\alpha)$, is defined as follows:

$$t(\alpha) = \begin{cases} f & \text{if } \alpha = \varepsilon, \\ t_i(\alpha') & \text{if } \alpha = i\alpha' \text{ and } 1 \leq i \leq n. \end{cases}$$

This gives a mapping from a sequence of integers to an alphabet in the tree. The α servers as a node designator.

The above function is obviously not defined for any α. Given a tree t, the set of all such valid α for t is called the *domain* of t.

Definition 2.5.2 (The domain for a tree).

$$\mathrm{Dom}(t) \stackrel{\text{def}}{=} \{\alpha | t(\alpha) \text{ is defined}\}.$$

Definition 2.5.3 (The domain of trees). Let F be a set of alphabets. We use $T^{\infty}(F)$ to denote the set of all trees over F and $T(F)$ to denote the set of all *finite* trees over F, that is of trees t whose domain $\mathrm{Dom}(t)$ is finite.

The set of all tress constitutes a complete metric space. This feature has useful applications, such as designating an infinite tree.

Definition 2.5.4 (The metric). The distance between two trees is defined by using the following auxiliary function δ:

$$\delta(t,t') \stackrel{\text{def}}{=} \begin{cases} \infty & \text{if } t = t' \\ \min\{|\alpha| | \alpha \in \mathrm{Dom}(t) \cap \mathrm{Dom}(t'), t(\alpha) \neq t(\alpha)\} & \text{if } t \neq t' \end{cases}$$

The distance d between trees t and t' is defined as follows:

$$d(t,t') \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } t = t' \\ 2^{-\delta(t,t')} & \text{if } t \neq t' \end{cases}$$

Proposition 2.5.1. *The followings are due to Courcelle [24].*

1. *The distance d makes the domain of trees $T^{\infty}(F)$ a complete metric space. Any Cauthy sequence of trees thus has the limit in $T^{\infty}(F)$.*
2. *Every uniformly continuous mapping: $T(F) \to E$ where E is complete extends uniquely into a uniformly continuous mapping: $T^{\infty}(F) \to E$.*

2.5.3 From a Value to its Infinite Expansion

Given a global heap $\mathbb{H}$ and a value v on a location p, then we say that tree t is the infinite expansion of v if t and v satisfy the relation $\mathbb{H} \models_p t : v$ defined by using the following rules:

$$\mathbb{H} \models_p () : ()$$

$$\mathbb{H} \models_p c : c$$

$$\mathbb{H} \models_p x : x$$

$$\mathbb{H} \models_p q : q$$

$$\frac{\mathbb{H} \models_p t : \mathbb{H}(p)(a)}{\mathbb{H} \models_p (\mathsf{ref}_{a@p} -)(t) : a}$$

$$\mathbb{H} \models_p P : P$$

$$\mathbb{H} \models_p u : u$$

$$\frac{\mathbb{H} \models_q t : \mathbb{H}(q)(a)}{\mathbb{H} \models_p (\mathsf{ref}_{a@q} -)(t) : a@q}$$

$$\frac{\mathbb{H} \models_p t : e}{\mathbb{H} \models_p (\lambda x.-)(t) : \lambda x.e}$$

$$\frac{\mathbb{H} \models_p t : e \quad \mathbb{H} \models_p t' : e'}{\mathbb{H} \models_p (--)(t,t') : ee'}$$

$$\frac{\mathbb{H} \models_p t : e}{\mathbb{H} \models_p (\mathsf{go}\ -)(t) : (\mathsf{go}\ e)}$$

$$\frac{\mathbb{H} \models_p t : e}{\mathbb{H} \models_p \langle - \rangle(t) : \langle e \rangle}$$

$$\frac{\mathbb{H} \models_p t : e \quad \mathbb{H} \models_p t' : e'}{\mathbb{H} \models_p (\mathsf{new}(-)-)(t,t') : \mathsf{new}(e)e'}$$

$$\frac{\mathbb{H} \models_p t : e}{\mathbb{H} \models_p (!-)(t) : !e}$$

$$\frac{\mathbb{H} \models_p t : e}{\mathbb{H} \models_p (?-)(t) : ?e}$$

$$\frac{\mathbb{H} \models_p t : e \quad \mathbb{H} \models_p t' : e' \quad \mathbb{H} \models_p t'' : e''}{\mathbb{H} \models_p (\mathsf{set}(-)--)(t,t',t'') : \mathsf{set}(e)e'e''}$$

Fixed $\mathbb{H}$, we wish to define the relation $\mathbb{H} \models_{_} - : -$ to be the greatest fixed point for the equation:

$$\mathbb{H} \models_{_} - : - = F_{\mathbb{H}}(\mathbb{H} \models_{_} - : -)$$

where $F_{\mathbb{H}}$ is an operator defined as follows.

$$\begin{aligned}
F_{\mathbb{H}}(Q) = \ & \{\langle p, (), () \rangle, \langle p, c, c \rangle, \langle p, x, x \rangle, \langle p, q, q \rangle, \langle p, P, P \rangle, \langle p, u, u \rangle\} \\
& \cup \{\langle p, (\mathsf{ref}_{a@p} -)(t), a \rangle | \langle p, t, \mathbb{H}(p)(a) \rangle \in Q\} \\
& \cup \{\langle p, (\mathsf{ref}_{a@q} -)(t), a@q \rangle | \langle q, t, \mathbb{H}(q)(a) \rangle \in Q\} \\
& \cup \{\langle p, (\lambda x.-)(t), \lambda x.e \rangle | \langle p, t, e \rangle \in Q\} \\
& \cup \{\langle p, (--)(t, t'), ee' \rangle | \langle p, t, e \rangle \in Q, \langle p, t', e' \rangle \in Q\} \\
& \cup \{\langle p, (\mathsf{go}\ -)(t), (\mathsf{go}\ e) \rangle | \langle p, t, e \rangle \in Q\} \\
& \cup \{\langle p, \langle - \rangle(t), \langle e \rangle \rangle | \langle p, t, e \rangle \in Q\} \\
& \cup \{\langle p, (\mathsf{new}(-)-)(t, t'), \mathsf{new}(e)e' \rangle | \langle p, t, e \rangle \in Q, \langle p, t', e' \rangle \in Q\} \\
& \cup \{\langle p, (!-)(t), !e \rangle | \langle p, t, e \rangle \in Q\} \\
& \cup \{\langle p, (?-)(t), ?e \rangle | \langle p, t, e \rangle \in Q\} \\
& \cup \{\langle p, (\mathsf{set}(-)--)(t, t', t''), \mathsf{set}(e)e'e'' \rangle | \ \langle p, t, e \rangle \in Q, \langle p, t', e' \rangle \in Q, \\
& \qquad\qquad \langle p, t'', e'' \rangle \in Q\}
\end{aligned}$$

This function $F_{\mathbb{H}}$ is *monotonic* in the sense that $Q \subseteq Q'$ implies $F_{\mathbb{H}}(Q) \subseteq F_{\mathbb{H}}(Q')$. The greatest fix point thus exists.

$$\mathbb{H} \models_{_} - : - \stackrel{\text{def}}{=} \bigcup \{Q | Q \subseteq F_{\mathbb{H}}(Q)\}.$$

This relation corresponds a value with its tree uniquely.

Proposition 2.5.2. *If $\mathbb{H} \models_p t : v$ and $\mathbb{H} \models_p t' : v$, then t and t' must be the same.*

Proof. By hypothesis, there is a relation Q such that $\langle p, t, v \rangle, \langle p, t', v \rangle \in Q$ and $Q \subseteq F_{\mathbb{H}}(Q)$. Suppose that t and t' are different. Then there must exist a node $f(t_1, ..., t_n)$ in t and a node $g(t'_1, ..., t'_m)$ in t' such that $\langle q, f(t_1, ..., t_n), v' \rangle, \langle q, g(t'_1, ..., t'_m), v' \rangle \in Q$ and $f \neq g$. This implies that $Q \not\subseteq F_{\mathbb{H}}(Q)$. The proof of $Q \not\subseteq F_{\mathbb{H}}(Q)$ is by induction on the structure of v'. □

Let us consider examples of infinite expansions of values. Suppose that $\mathbb{H} = \{p \mapsto \{a \mapsto (\mathbf{copy}, b), b \mapsto (\mathbf{copy}, a), c \mapsto (\mathbf{copy}, 2)\}\}$. Then we have that:

$$\mathbb{H} \models_p (\mathsf{ref}_{c@p} -)(2) : c, \qquad \mathbb{H} \models_p (\mathsf{ref}_{a@p} -)((\mathsf{ref}_{b@p} -)(...)) : a, \text{and}$$

$$\mathbb{H} \models_p (\mathsf{ref}_{b@p} -)((\mathsf{ref}_{a@p} -)(...)) : b.$$

The infinite expansion of a may be written as $\mathsf{ref}_{a@p}(\mathsf{ref}_{b@p}(...))$ or $\mu t.\mathsf{ref}_{a@p}(\mathsf{ref}_{b@p}(t))$ by using the fixed point operator μ.

The infinite expansion is invariant according to extending the global heap.

Proposition 2.5.3. *If $\mathbb{H} \models_p t : v$ and $\mathbb{H} \sqsubseteq \mathbb{H}'$ then $\mathbb{H}' \models_p t : v$.*

Proof. Because $Q \subseteq F_{\mathbb{H}}(Q)$ implies $Q \subseteq F_{\mathbb{H}'}(Q)$. □

2.5.4 Equivalence between Values in the Heaps

On migration, a set of values is copied to the destination heap, on the newly allocated addresses. To state that the transmitted data structure is preserved on migration, the values in the departure heap must be compared with the values in the destination heap. But in case of comparing their addresses, the addresses in the destination heap can be allocated newly and arbitrarily. Therefore the correspondence between old addresses and new ones must be maintained.

Let the departure heap $H = \{a \mapsto (\mathbf{copy}, b), b \mapsto (\mathbf{copy}, a)\}$, the destination heap $H' = \{c \mapsto (\mathbf{copy}, d), d \mapsto (\mathbf{copy}, c)\}$ and the global heap $\mathbb{H} = \{p \mapsto H, q \mapsto H'\}$. Our intuition on migration behind these heaps is that address a at location p is moved to address c at location q and that address b at location p is moved to address d at location q. Intuitively, the data structure in H is equivalent to the one in H' (that is, a mutual recursion). The correspondence between addresses in the different heaps is represented by a finite set of pairs of corresponding addresses, denoted by Δ. In this case, Δ is $\{(a, c), (b, d)\}$.

The Δ must be functional and injective. The reason why Δ is functional is that a value is duplicated to the destination heap just once for a single migration. Therefore if $(a, b), (a, c) \in \Delta$ then $b = c$. The reason why Δ is injective is that otherwise sharing is violated. Suppose that $H' = \{c \mapsto (\mathbf{copy}, c)\}$ and $\Delta = \{(a, c), (b, c)\}$, meaning that both addresses a and b are considered as single address c at location q. In this case, their data structures seems to be equivalent (that is, $\mu t.\mathsf{ref}_{a@p}(\mathsf{ref}_{b@p}(t))$ and $\mu t.\mathsf{ref}_{c@q}(t)$) but the result of updating address a is different from the one of updating address c.

The equivalence relation $\simeq$ on trees is thus defined as follows.

Definition 2.5.5 (Equivalence between infinite expansions).

$$t \simeq_\Delta t' \stackrel{\text{def}}{=} \mathrm{Dom}(t) = \mathrm{Dom}(t') \wedge \bigwedge_{\alpha \in \mathrm{Dom}(t)} \begin{cases} \text{if } t(\alpha) = \mathsf{ref}_{a@p} - \wedge (a, b) \in \Delta \text{ then } t'(\alpha) = \mathsf{ref}_{b@q} - \\ \text{if } t(\alpha) = \mathsf{ref}_{a@p} - \wedge a \notin \mathrm{Dom}(\Delta) \text{ then } t'(\alpha) = \mathsf{ref}_{a@p} - \\ \text{otherwise } t(\alpha) = t'(\alpha) \end{cases}$$

Let us take an example. Suppose that $\mathbb{H} = \{p \mapsto \{a \mapsto (\mathbf{copy}, b), b \mapsto (\mathbf{copy}, a)\}, q \mapsto \{c \mapsto (\mathbf{copy}, d), d \mapsto (\mathbf{copy}, c)\}\}$. The infinite expansion of a is $\mu t.\mathsf{ref}_{a@p}(\mathsf{ref}_{b@p}(t))$ because $\mathbb{H} \models_p \mu t.\mathsf{ref}_{a@p}(\mathsf{ref}_{b@p}(t)) : a$. Similarly, the infinite expansion of c is $\mu t.\mathsf{ref}_{c@p}(\mathsf{ref}_{d@p}(t))$. Then we have that:

$$\mu t.\mathsf{ref}_{a@p}(\mathsf{ref}_{b@p}(t)) \simeq_\Delta \mu t.\mathsf{ref}_{c@p}(\mathsf{ref}_{d@p}(t))$$

where $\Delta = \{(a, c), (b, d)\}$.

We wish to define the equivalence between the departure *heap* and the destination *heap* by using the equivalence between infinite expansions. In general, a heap contains the values not related to migration. Therefore the values to be considered for the equivalence are restricted to those related to migration.

Definition 2.5.6 (Equivalence between the heaps). Let $\mathbb{H}$ be the global store before migration takes place and $\mathbb{H}'$ be the global store after migration took place. We say that $\mathbb{H}$ and $\mathbb{H}'$ are *equivalent* if the moved values are equivalent to the original values.

$$\frac{\mathbb{H} \models_p t : a \quad \mathbb{H}' \models_q t' : a' \quad t \simeq_\Delta t' \quad \text{for all } (a, a') \in \Delta}{\mathbb{H} \simeq_{\Delta, p, q} \mathbb{H}'}$$

2.5.5 Soundness and Completeness

In this subsection, we show various formal properties of the movement relation $\Rightarrow$.

Lemma 2.5.1 (Extension). *If* $\sigma, \Delta, H_d, e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'$, *then we have that* $\sigma \sqsubseteq \sigma'$, $\Delta \sqsubseteq \Delta'$, *and* $H_d \sqsubseteq H'_d$.

Proof. The proof is by induction on the derivation of $\Rightarrow_{H,p,p'}$.
Case (Move Copy) By induction hypothesis, we have that $\sigma \sqsubseteq \sigma'$, $\Delta \uplus \{(a, a')\} \sqsubseteq \Delta'$, and $H_d \uplus \{a' \mapsto (\mathbf{copy}, v')\} \sqsubseteq H'_d$. From $a \notin \mathrm{Dom}(\Delta)$, $\Delta \sqsubseteq \Delta \uplus \{(a, a')\}$. From the transitivity, $\Delta \sqsubseteq \Delta'$.
Case (Move Resident) From the reflexivity, we have that $\sigma \sqsubseteq \sigma$, $\Delta \sqsubseteq \Delta$, and $H_d \sqsubseteq H_d$.
Case (Move Carry) By induction hypothesis, we have that $\Delta \uplus \{(a, a')\} \sqsubseteq \Delta'$, and $H_d \uplus \{a' \mapsto (\mathbf{carry}, v')\} \sqsubseteq H'_d$. From $a \notin \mathrm{Dom}(\Delta)$, $\Delta \sqsubseteq \Delta \uplus \{(a, a')\}$. From $a' \notin \mathrm{Dom}(H_d)$, $H_d \sqsubseteq H_d \uplus \{a' \mapsto (\mathbf{carry}, v')\}$.
Otherwise The other cases can be proved similarly. □

Lemma 2.5.2 (Invariant conditions). *Suppose that* $\sigma, \Delta, H_d, e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'$, *then we have that*

1. *if* $\sigma \subseteq \Delta$ *then* $\sigma' \subseteq \Delta'$,
2. *if* $Dom(\Delta) \subseteq Dom(H)$ *then* $Dom(\Delta') \subseteq Dom(H)$, *and*
3. $Rng(\Delta) = Dom(H_d)$ *iff* $Rng(\Delta') = Dom(H'_d)$.

Proof sketch. The proof is by induction on the derivation of $\Rightarrow_{H,p,p'}$. □

Theorem 2.5.1 (Transparency for data transfer). *Let* $\mathbb{H} = \mathbb{H}_0 \uplus \{p \mapsto \sigma H_p, p' \mapsto H_{p'} \uplus H_d\}$ *for some* $\mathbb{H}_0$. *Suppose that* $\sigma, \Delta, H_d, e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'$, $\mathbb{H} \models_p t_e : e$, *and* $\forall a \in Dom(\sigma H_p) \exists t \mathbb{H} \models_p t : \sigma H_p(a)$. *Then we have that:*

1. $\mathbb{H}' \models_{p'} t'_e : e'$ *for some* t'_e,
2. $t_e \simeq_{\Delta'} t'_e$, *and*
3. $\mathbb{H} \simeq_{\Delta',p,p'} \mathbb{H}'$

where $\mathbb{H}' = \mathbb{H}_0 \uplus \{p \mapsto \sigma' H_p, p' \mapsto H_{p'} \uplus H'_d\}$.

Proof. The proof is by induction on the derivation of $\sigma, \Delta, H_d, e \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, e'$. The essentially important cases are (Move Copy), (Move Resident) and (Move Carry).
Case (Move Copy)

$$\frac{H(a) = (\mathbf{copy}, v) \quad \sigma, \Delta \uplus \{(a, a')\}, H_d \uplus \{a' \mapsto (\mathbf{copy}, v')\}, v \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, v'}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, a'}$$

The induction hypothesis is that if

$$\sigma, \Delta \uplus \{(a, a')\}, H_d \uplus \{a' \mapsto (\mathbf{copy}, v')\}, v \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, v', \tag{2.1}$$

$$\mathbb{H}_I \models_p t_I : v, \tag{2.2}$$

$$\forall a \in \mathrm{Dom}(\sigma H_p) \exists t \mathbb{H}_I \models_p t : \sigma H_p(a) \tag{2.3}$$

then we have that

$$\mathbb{H}' \models_{p'} t'_I : v', \tag{2.4}$$
$$t_I \simeq_{\Delta'} t'_I, \tag{2.5}$$
$$\mathbb{H}_I \simeq_{\Delta',p,p'} \mathbb{H}' \tag{2.6}$$

where $\mathbb{H}_I = \mathbb{H}_0 \uplus \{p \mapsto \sigma H_p, p' \mapsto H_{p'} \uplus H_d \uplus \{a' \mapsto (\mathbf{copy}, v')\}\}$.

By (Move Copy), we have (2.1). By hypothesis, there exists t_e such that $\mathbb{H} \models_p t_e : a$. Thus, we must have

$$\frac{\mathbb{H} \models_p t_v : v}{\mathbb{H} \models_p t_e : a}$$

for some t_v. By Proposition 2.5.2, $t_v = t_I$. Since $\mathbb{H} \sqsubseteq \mathbb{H}_I$, we have $\mathbb{H}_I \models_p t_I : v$ by Proposition 2.5.3. By hypothesis, $\forall a \in \mathrm{Dom}(\sigma H_p) \exists t \mathbb{H} \models_p t : \sigma H_p(a)$. Since $\mathbb{H} \sqsubseteq \mathbb{H}_I$, we also have $\forall a \in \mathrm{Dom}(\sigma H_p) \exists t \mathbb{H}_I \models_p t : \sigma H_p(a)$ by Proposition 2.5.3. As a result, we have (2.4), (2.5) and (2.6).

From (2.4) and the definition of $\models$, we have $\mathbb{H}' \models_{p'} \mathsf{ref}_{a'@p'} t'_I : a'$.

$$\begin{aligned} t_e \simeq_{\Delta'} t'_e &\Leftrightarrow \mathsf{ref}_{a@p} t_I \simeq_{\Delta'} \mathsf{ref}_{a'@p'} t'_I \\ &\Leftrightarrow (a, a') \in \Delta' \wedge t_I \simeq_{\Delta'} t'_I \end{aligned}$$

Since $\Delta \uplus \{(a, a')\} \sqsubseteq \Delta'$, $(a, a') \in \Delta'$ from Lemma 2.5.1. Since $t_I \simeq_{\Delta'} t'_I$ from (2.5), we have $t_e \simeq_{\Delta'} t'_e$. From (2.6), we have $\mathbb{H}_I \simeq_{\Delta',p,p'} \mathbb{H}'$. Since $\Delta' \subseteq \mathrm{Dom}(\sigma H)$ from Lemma 2.5.1, we also have $\mathbb{H} \simeq_{\Delta',p,p'} \mathbb{H}'$.

Case (Move Resident)

$$\frac{H(a) = (\mathbf{resident}, v)}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, a@p}$$

By hypothesis, we have that

$$\frac{\mathbb{H} \models_p t_v : v}{\mathbb{H} \models_p \mathsf{ref}_{a@p} t_v : a}$$

for some t_v. By the definition of $\models$, we have that

$$\frac{\mathbb{H} \models_p t_v : v}{\mathbb{H} \models_{p'} \mathsf{ref}_{a@p} t_v : a@p}$$

Since $t_e = t'_e = \mathsf{ref}_{a@p} t_v$, $t_e \simeq_{\Delta'} t'_e$. Since $\mathbb{H} = \mathbb{H}'$, $\mathbb{H} \simeq_{\Delta',p,p'} \mathbb{H}'$.

Case (Move Carry)

$$\frac{H(a) = (\mathbf{carry}, v) \qquad \sigma \uplus \{(a, a')\}, \Delta \uplus \{(a, a')\}, H_d \uplus \{a' \mapsto (\mathbf{carry}, v')\}, v \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, v'}{\sigma, \Delta, H_d, a \Rightarrow_{H,p,p'} \sigma', \Delta', H'_d, a'}$$

The proof is almost the same as the case of (Move Copy). The important difference is the fact that $\mathbb{H}_I$ is defined as $\mathbb{H}_I = \mathbb{H}_0 \uplus \{p \mapsto (\sigma \uplus \{(a, a')\}) H_p, p' \mapsto H_{p'} \uplus H_d \uplus \{a' \mapsto (\mathbf{carry}, v')\}\}$. So, $\mathbb{H}_I$ is *not* an extension of $\mathbb{H}$ ($\mathbb{H} \not\sqsubseteq \mathbb{H}_I$). We must thus show $\mathbb{H} \simeq_{\Delta',p,p'} \mathbb{H}'$ without using Proposition 2.5.3.

By the definition of $\simeq$,

$$(\mathbb{H}_I \simeq_{\Delta',p,p'} \mathbb{H}' \rightarrow \mathbb{H} \simeq_{\Delta',p,p'} \mathbb{H}') \Leftrightarrow (\mathbb{H}_I \models_p t_I : a, t_I \simeq_{\Delta'} t' \rightarrow \mathbb{H} \models_p t : a, t \simeq_{\Delta'} t')$$

By hypothesis, there exist t_v and t'_v such that $\mathbb{H} \models_p t_v : v$ and $\mathbb{H}' \models_{p'} t'_v : v'$. From the definition of $\mathbb{H}_I$, t_I is the tree equivalent to t except that every occurrence of $\mathsf{ref}_{a@p} t_v$ is replaced with $\mathsf{ref}_{a'@p'} t'_v$. Therefore,

$$\begin{aligned}(\mathbb{H}_I \models_p t_I : a, t_I \simeq_{\Delta'} t' \to \mathbb{H} \models_p t : a, t \simeq_{\Delta'} t') &\Leftrightarrow \mathsf{ref}_{a@p} t_v \simeq_{\Delta'} \mathsf{ref}_{a'@p'} t'_v \\ &\Leftrightarrow (a, a') \in \Delta' \wedge t_v \simeq_{\Delta'} t'_v\end{aligned}$$

From Lemma 2.5.1, $(a, a') \in \Delta'$. By induction hypothesis, $t_v \simeq_{\Delta'} t'_v$. □

Corollary 2.5.1. *Let $\mathbb{H} = \mathbb{H}_0 \uplus \{p \mapsto H_p, p' \mapsto H_{p'}\}$ for some $\mathbb{H}_0$. Suppose that $\emptyset, \emptyset, \emptyset, e \Rightarrow_{H,p,p'} \sigma, \Delta, H_d, e'$, $\mathbb{H} \models_p t_e : e$, and $\forall a \in Dom(H_p) \exists t \mathbb{H} \models_p t : H_p(a)$. Then we have $\mathbb{H} \simeq_{\Delta,p,p'} \mathbb{H}'$ where $\mathbb{H}' = \mathbb{H}_0 \uplus \{p \mapsto \sigma H_p, p' \mapsto H_{p'} \uplus H_d\}$.*

Next, let us discuss the topic of the completeness.

Lemma 2.5.3 (Uniqueness). *If two expressions e and e' have the equivalent infinite expansions (e.g. $\mathbb{H} \models_p t : e$, $\mathbb{H} \models_{p'} t' : e'$ and $t \simeq_{\Delta'} t'$), we must have $e' \equiv \theta e$ for substitution θ such that:*

1. *θ is the minimal substitution over local and remote addresses satisfying the following conditions.*
2. *If $a \in Dom(\Delta)$, then $\theta a = \Delta(a)$.*
3. *If $a \in Dom(\mathbb{H}(p))$ and $a \notin Dom(\Delta)$, then $\theta a = a@p$.*
4. *$\theta(a'@p') = a'$ for any $a \in Dom(\mathbb{H}'(p'))$.*

Proof. The proof is by induction on the structure of e.
Case $(e = c)$ By the definition of $\models$, we must have $t = c$. By the definition of $\simeq$, we must have $t' = c$. By the definition of $\models$, we must have $e' = c$. Obviously, $\theta c \equiv c$.
Case $(e = a)$ By the definition of $\models$, $t = \mathsf{ref}_{a@p} t_v$ for some t_v. Suppose that $(a, a') \in \Delta$. By the definition of $\simeq$, we must have $t' = \mathsf{ref}_{a'@p'} t'_v$ for some t'_v. By the definition of $\models$, we must have either $\mathbb{H}' \models_{p'} t' : a'$ or $\mathbb{H}' \models_{p'} t' : a'@p'$, but the latter case cannot happen because $a'@p'$ is not a valid value in location p'. So, $e' \equiv a'$. Thus we have $\theta a = a' = \Delta(a)$. On the other hand, suppose that $(a, a') \notin \Delta$. By the definition of $\simeq$, we must have $t' = \mathsf{ref}_{a@p} t'_v$ for some t'_v. By the definition of $\models$, we must have $\mathbb{H}' \models_{p'} \mathsf{ref}_{a@p} t'_v : a@p$. Thus we have $\theta a = a@p$.
Case $(e = a''@p')$ From the definition of $\models$, we must have $\mathbb{H} \models_p \mathsf{ref}_{a''@p'} t_v : a''@p'$ for some t_v. Because a'' is a local address in location p', $a'' \notin \mathrm{Dom}(\Delta)$. Therefore we must have $t' = \mathsf{ref}_{a''@p'} t_v$ by the definition of $\simeq$. From the definition of $\models$, we must have $\mathbb{H}' \models_{p'} \mathsf{ref}_{a''@p'} t_v : a''$. Thus we have $\theta(a''@p') = a''$.
Case $(e = a''@p'', p'' \neq p, p'' \neq p')$ From the definition of $\models$, we must have $\mathbb{H} \models_p \mathsf{ref}_{a''@p''} t_v : a''@p''$ for some t_v. From the definition of $\simeq$, we must have $t' = \mathsf{ref}_{a''@p''} t'_v$ for some t'_v. From the definition of $\models$, we must have $e' = a''@p''$. Thus we have $\theta(a''@p'') = a''@p''$.
Otherwise The other cases are simply proved by applying induction. □

Lemma 2.5.4 (Existence). *Let $\mathbb{H} = \mathbb{H}_0 \uplus \{p \mapsto H_1, p' \mapsto H_2\}$. Given e, e' and $\mathbb{H}' = \mathbb{H}_0 \uplus \{p \mapsto \sigma' H'_1, p' \mapsto H'_2\}$, suppose that there are t, t' and Δ' such that $\mathbb{H} \models_p t : e$, $\mathbb{H}' \models_{p'} t' : e'$ and $t \simeq_{\Delta'} t'$. Then there exist:*

- *Δ° such that $\Delta \subseteq \Delta^\circ \subseteq \Delta'$,*

- σ° *such that* $\sigma^\circ = \Delta^\circ \cap \sigma'$,
- $\sigma \subseteq \Delta \cap \sigma'$,
- H_d° *such that* $H_d \sqsubseteq H_d^\circ$ *and* $Dom(H_d^\circ) = Rng(\Delta^\circ)$,
- $H_d^\dagger$ *is equivalent to* H_d *except its constituent data movement protocols, and*
- $H_p^\dagger$ *is equivalent to* H_1 *except its constituent data movement protocols: when* $a \notin Dom(\Delta')$, *the data movement protocol is resident type; when* $a \in Dom(\Delta') \cap Dom(\sigma')$, *the data movement protocol is carry type; otherwise, the data movement protocol is copy type.*

such that $\sigma, \Delta, H_d^\dagger, e \Rightarrow_{H_p^\dagger, p, p'} \sigma^\circ, \Delta^\circ, H_d^\circ, e'$ *where* H_d *and* Δ *are defined as follows:*

- H_d *: the subset of* H_2 *such that* $Dom(H_d) = Rng(\Delta')$, *and*
- Δ *: the subset of* Δ' *such that* $Rng(\Delta) = Dom(H_d)$.

Proof. The proof is by induction on the lexicographic order $<$ on the pair of the size of $\Delta' \setminus \Delta$ and the structure of e. We have $(n, e) < (n', e')$ if and only if $n < n'$ or, $n = n'$ and e is a subexpression of e'. The essentially important cases are the ones where e is a local address.

Case $(|\Delta' \setminus \Delta| = 0, e = a)$ By hypothesis, $\Delta = \Delta^\circ = \Delta'$. By the definition of $\models$, we must have $t = \mathsf{ref}_{a@p} t_v$ for some t_v. Let us examine the case where $(a, a') \in \Delta'$. By the definition of $\simeq$, we have that $t' = \mathsf{ref}_{a'@p'} t'_v$ and $t_v \simeq_{\Delta'} t'_v$ for some t'_v. By the definition of $\models$, we have $e' = a'$. Let $(P', v') = \mathbb{H}'(p')(a')$. We have $\sigma', \Delta', H_d^\dagger, e \Rightarrow_{H_p^\dagger, p, p'} \sigma', \Delta', H_d^\dagger, e'$ by (Move Addr). Next, consider the case where $(a, a') \notin \Delta'$. By the definition of $\simeq$, we must have that $t' = \mathsf{ref}_{a@p} t'_v$ and $t_v \simeq_{\Delta'} t'_v$ for some t'_v. By the definition of $\models$, we have $e = a@p$. The data movement protocol of a must be resident protocol. We have $\sigma', \Delta', H_d^\dagger, e \Rightarrow_{H_p^\dagger, p, p'} \sigma', \Delta', H_d^\dagger, e'$ by (Move Resident).

Case $(|\Delta' \setminus \Delta| > 0, e = a)$ Consider the case where $(a, a') \in \Delta'$. By the definition of $\models$, we have $t = \mathsf{ref}_{a@p} t_v$ for some t_v. Since $\mathsf{ref}_{a@p} t_v \simeq_{\Delta'} t'$, we have that $t' = \mathsf{ref}_{a'@p'} t'_v$ and $t_v \simeq_{\Delta'} t'_v$ for some t'_v. By the definition of $\models$, we have $e' = a'$. Let $(P', v') = \mathbb{H}'(p')(a')$. If $(a, a') \in \Delta$, taking $\Delta^\circ = \Delta$ and $H_d^\circ = H_d^\dagger$, we have $\sigma', \Delta, H_d^\dagger, e \Rightarrow_{H_p^\dagger, p, p'} \sigma', \Delta, H_d^\dagger, e'$ by (Move Addr). In the case where $(a, a') \notin \Delta$, let us define $\mathbb{H}^\dagger$ to be $\mathbb{H}_0 \uplus \{p \mapsto H_1, p' \mapsto H_2 \uplus \{a' \mapsto (P', v')\}\}$. Since $\mathbb{H} \sqsubseteq \mathbb{H}^\dagger$, we also have that $\mathbb{H}^\dagger \models_p t^\dagger : e$ and $t^\dagger \simeq_{\Delta'} t'$. And the consider the proposition to be proved by using $\mathbb{H}^\dagger$ instead of $\mathbb{H}$. Since this decreases the size of $\Delta' \setminus \Delta$, by induction hypothesis, we have that $\sigma, \Delta \uplus \{(a, a')\}, H_d^\dagger \uplus \{a' \mapsto (P', v')\}\}, v \Rightarrow_{H_p^\dagger, p, p'} \sigma^\circ, \Delta^\circ, H_d^\circ, v'$. If $(a, a') \in \sigma$, then we have $\sigma, \Delta, H_d^\dagger, a \Rightarrow_{H_p^\dagger, p, p'} \sigma^\circ, \Delta^\circ, H_d^\circ, a'$ by (Move Carry). Otherwise, we have $\sigma, \Delta, H_d^\dagger, a \Rightarrow_{H_p^\dagger, p, p'} \sigma^\circ, \Delta^\circ, H_d^\circ, a'$ by (Move Copy). Next, let us consider the case where $(a, a') \notin \Delta'$. In this case, e' must be $a@p$. So, the relation holds by (Move Resident). □

Theorem 2.5.2 (Completeness of data movement protocols). *Let* $\mathbb{H} = \mathbb{H}_0 \uplus \{p \mapsto H_p, p' \mapsto H_{p'}\}$ *for some* $\mathbb{H}_0$. *Suppose that* $\mathbb{H} \simeq_{\Delta, p, p'} \mathbb{H}'$ *for some* Δ *and* $\mathbb{H}' = \mathbb{H}_0 \uplus \{p \mapsto \sigma H_p, p' \mapsto H_2\}$ *such that* $Dom(H_{p'}) \cap Dom(\Delta) = \emptyset$, *then there exists an expression* e *such that* $\emptyset, \emptyset, \emptyset, e \Rightarrow_{H_p, p, p'} \sigma, \Delta, H_d, e'$ *and* H_d *is a subset of* H_2 *ignoring the difference on data movement protocol.*

Proof. Take e as an expression $((\lambda x_1...x_n.())a_1...a_n)$ where $\{a_1, ..., a_n\} = \mathrm{Dom}(\Delta')$. From Lemma 2.5.4, we have $\emptyset, \emptyset, \emptyset, e \Rightarrow_{H_p,p,p'} \sigma, \Delta, H_d, e'$. From Corollary 2.5.1, $\mathbb{H} \simeq_{\Delta,p,p'} \mathbb{H}_1$ where Let $\mathbb{H}_1 = \mathbb{H}_0 \uplus \{p \mapsto \sigma H_p, p' \mapsto H_{p'} \uplus H_d\}$. From Lemma 2.5.3, when we have $\mathbb{H} \simeq_{\Delta,p,p'} \mathbb{H}'$ and $\mathbb{H} \simeq_{\Delta,p,p'} \mathbb{H}_1$, we have $v = v'$ where $\mathbb{H}'(p')(a) = (P, v)$, $\mathbb{H}_1(p')(a) = (P', v')$ for all $a \in \mathrm{Rng}(\Delta)$. Since $\mathrm{Rng}(\Delta) = \mathrm{Dom}(H_d)$, H_d is a subset of H_2 ignoring the difference on data movement protocol. □

2.6 Related Work

2.6.1 Core Facile

Core Facile [4] is much similar to our calculus. Both have evaluation contexts and parallel composition. The difference can be summarized as follows.

- There is no notion of location in Core Facile.
- In Core Facile, parallel composition can be composed within sequential constructs. For instance, $\lambda x.(Px|Q)$ is a valid term in Core Facile.
- Core Facile has communication channels as primitive.
- There is no notion of heaps in Core Facile.
- Core Facile has equational theories.

We think that the analysis carried out in the thesis can be applicable on Core Facile if Core Facile is extended to possess the notion of heaps.

There are other semantic frameworks which have both sequential and concurrent execution. A semantics for concurrent ML [11] is constructed on a concurrent calculus. The sequential part of a program is translated into a term in the calculus, while preserving the semantics.

2.6.2 A Model of Distributed Memory Management

Ungureanu proposes a formal syntactic model of distributed memory management [95, 96] as an extension of Morriset's models of memory management [72] for distributed environment. The model of distributed memory management is based on a transition system between global environments as our operational semantics. A global environment means the state of all distributed heaps. The notion of flat location is adopted in the model. Each location has a sequential program (represented by a lambda expression) whose evaluation order is defined by evaluation contexts. There is no notion of remote references, but communication channels are available by which values can be transmitted between different locations. An observation equivalence relation is defined and formal definitions for two different garbage collections are given.

The major problem in applying the model for mobile computation is that it lacks the notion of migration. As we have shown, the value transmission on migration in existing mobile systems is not a simple duplication. To formalize the mechanism of the value transmission, a mechanism like the *data movement relation* would be needed.

2.6.3 Mobile Calculi as Extensions of Concurrent Calculi

There are several variants of mobile calculi that bear a close resemblance to π-calculus [69], such as distributed π-calculus [85], π_l-calculus [6], Dπ [82] and others [51, 81]. In addition, there are a calculus of hierarchical locations called *mobile ambient* [19] and a distributed extension [36] of the join calculus [37].

These calculi are suitable for discussing abstract notions in mobile computation, but it is difficult for them to target existing mobile language systems. The reason can be summarized as follows.

- They are not suitable to discuss the control in sequential programming language. (For instance, what part of code is to be moved on migration.) Because the notion of control is *ambiguous* in concurrent calculi as pointed out in Refs. [52, 71, 79].
- The major difference in existing mobile language systems appears on the point of the way the heaps are modified on migration. Their mobile calculi, however, lack the notion of heaps.

These problems are the very point we develop in the thesis work. Therefore, we think that our work complements exactly these unsatisfactory points of mobile calculi.

2.7 Features not Supported in the Calculus

This section describes what is *not* supported in our mobile calculus. The reason why they are not supported is that we consider that they should be approached in a different way.

2.7.1 Module Mechanism and Variable-Binding

A *module* is a term to designate the unit of linking [18]. The module mechanism is closely related to mobile computation because a mobile program execution can be considered as a module. A mobile program execution is a partial collection of code and data. It is dynamically loaded into the memory, meaning that the migrated code and data are allocated appropriate memory areas and the internal and external references are re-connected to their appropriate addresses in the memory. This is the very mechanism of dynamic linking. A mobile program execution is more precisely a *functor* [70] in the sense that it receives the runtime environment and becomes a module.

This perspective raises a problem "*what is the sophisticated module mechanism for mobile programming languages?*" This problem appears in the language level as a problem of assigning the correct environment for a symbol or a variable. Because a program is migrated between different hosts and each host has its own environment, the environment to which a variable belongs is in general ambiguous. Cardelli [16] gives a solution to this problem such that the environment is completely determined by the lexical scope. Later, another solution is proposed as the ambient calculus [19], where some operators are provided that control environments. Dean discusses the relation between static typing and dynamic linking [30]. Acharya proposes a module mechanism in which the environment of a symbol is determined by the attribute assigned to the symbol [2], which is especially designed for mobile programming

languages. The naming resolution is the critical mechanism in CORBA. In general, a calculus with first-class environments [75] and context calculi [27, 48] would be useful for the problem.

2.7.2 Hierarchical Locations

The notion of location adopted in our calculus is flat, but many other mobile calculi [19, 82, 85] and mobile systems such as Telescript [41] have hierarchical location. From the practical view point, there are several bases to support hierarchical locations. For instance, when the programmer wants to provide several services in the same host, he/she can allocate a location for each service as a sub-location of the host. This separate allocation would alleviate the cost of administration of the services. Henessy proposes a mobile calculus [51] where the location is considered to be flat. The emphasis of the calculus is to represent resource (something local to a location) access control. In our calculus, there is something local, namely a heap. It is convenient to consider in the setting of flat location when we want to concentrate on analysis of something local.

2.7.3 General Pointer Forwarding

Our calculus does not provide pointer forwarding mechanism per se, but pointer forwarding is available only in a restricted form: when a value of carry or takeaway protocol is moved to a different host, the references to the value are forwarded to refer the moved (fresh) reference only in the departure host. If there is a reference to the original value in another host, it is not forwarded, but still refers to the original value. The reason why our calculus does not provide a global pointer forwarding mechanism is that it is not trivial and raises troublesome implementational problems.

Let us consider the case where there are a single shared value of carry protocol and several remote references to the value in different locations. If the shared value is moved, that information must be broadcasted to bring about pointer forwarding: each remote reference to the value must be replaced with the new one. But what will happens if the original value is updated *before* the broadcasted information arrives? To faithfully follow the semantics, we must develop complicated protocols. This also applies to implementation of channels of concurrent calculi. When a value is sent to a channel, a receiver gets the value wherever it is, which also requires complicated protocols.

In the context of non-mobile concurrent calculi, these problems are separated from the calculi and are considered as implementational problems. On the other hand, in the context of mobile computations, we cannot avoid these problems, or rather, it is a merit of mobile calculus to describe such implementational problems.

Sewell presents an agent calculus [86] based on the distributed π-calculus [85]. In the agent calculus, location-independence is guaranteed, meaning that the result of a location-independent operation does not depend on the location where the operation is performed. For instance, transmission of a message along with a channel is location-independent. The receiver surely receives a message wherever the sender and the receiver are located. Note that location-independence of message transmission is not realized by nature in the low level, such as TCP/IP. He gives an encoding from the agent calculus to the distributed π-calculus and it turns out that the encoding is not simple.

Chapter 3

A Programming Language for Mobile Computation

A programming language based on our mobile calculus is presented. We describe the features of the language for mobile computation as an extension, modification of an object-oriented language like C++ and Java. The programmer can describe flexible migration by using only three language primitives (*go*, *undock* and *migratory*) and the data movement protocols.

We aim to exploit existing software resources as much as possible for mobile computation. From this standpoint, it is important that migration is transparent and is completely controlled by only a few programming constructs. In a fortunate case, existing program can be mobile by adding only a single go statement in our mobile language. On the contrary, an existing mobile language usually requires restructuring for a non-mobile program to work correctly on migration.

3.1 Initiating Migration by Using Go Primitive

Migration takes place by executing a "go" statement. Suppose a method of some object has the following lines of code:

```
printf ("bye!");
go ("//ritsuko:2001/Executor");
printf ("nice to meet you!");
```

The argument of a go statement is the name of a migration server registered in the object request broker (ORB) or the JavaRMI registry. The migration server is an object that manages migration. When the go statement is executed, the execution state and object are transmitted to the destination host, where the object continues executing just after the go statement. In the above example, `"bye!"` is displayed on the departure host, and `"nice to meet you!"` is displayed on the destination host (ritsuko).

The migration mechanism in our system is transparent. (1) Even in cases where a go statement appears in a compound statement, such as a for statement, try statement, switch statement, or if statement, the migrated program resumes executing at the destination host. (2) If callee method `foo` including a go statement is invoked by caller method `bar`, control returns to method `bar` on the *destination* host after the method `foo` returns. This behavior is quite different from that of remote evaluation [89], remote procedure call (RPC) and remote method invocation (RMI). For instance, a migratory Fibonacci method can be written in our language as follows:

```
bool Moved = false;
migratory int fib( int n ) {
    if ( n == 0 ) {
        if ( !Moved ) {
            go ("//aki:2001/Executor");
            Moved = true;
        }
        return 1;
    }
    else if ( n == 1 )
        return 1;
    else
        return fib (n-1) + fib (n-2);
}
```

Method `fib` computes a Fibonacci number by recursively invoking itself. During the computation, `fib` (and the object executing it) migrates only once (when n is equal to 0), that is, at the point where the execution stack is the deepest. Many systems (e.g., those described in Refs. [64] and [76]) that support mobile computation do not allow the transmission of recursive function invocations. The programmer must therefore write a method and auxiliary class definitions that represent the rest of the computation explicitly. (When we wrote a Fibonacci method in Voyager [76], we had to add extra 35 lines and a class definition representing the execution state.) The keyword `migratory` at the head of the method above is explained in Section 3.3.

3.2 Controlling Transparency by Using Undock Primitive

An undock statement serves as a *marker* that specifies the range of the area to be migrated in the execution stack. It makes it possible to control migration transparency. The brackets of the undock statement restrict the effect of a go statement to the enclosed statements. When the go statement in the undock statement is executed, the rest after the go statement in the undock statement resumes executing at the destination host, while the statements after the undock statement are concurrently executed on the departure host.

```
undock {
    go ("//ritsuko:2001/JavaGoExecutor");
    printf ("nice to meet you!");
}
printf ("bye!");
```

In the above lines of code, only the statements enclosed in the `undock` brackets are migrated. Therefore, `"bye!"` is displayed on the departure host, not on the destination host. An undock statement has a dynamic extent similar to a try-and-catch statement. When a go statement is executed in the nested extents of several undock statements, the migrating part is the inside area of the most recently executed undock statement. An area to migrate can be specified by an undock statement in any method defined in any class.

Figure 3.1 depicts the migrating areas in a stack where methods `a`, `b`, `c`, and `d` are called in this order. The stack grows upwards according to a method invocation.

```
o4.d() { ...  go(...)  ... }
o3.c() { ...  o4.d() ... }
o2.b() { ...  undock { ...  o3.c() ... } ...  }
o1.a() { ...  o2.b() ...  }
```

Figure 3.1: Conceptual diagram of a stack.

The $o.m()$ denotes the invocation of method m of object o. The portions underlined show the migrating areas. The stack frame of the `o1.a()` is not migrated because the execution of the go statement in method `d` is in the extent of the undock statement in method `b`. The stack frames corresponding to the migrating methods are transmitted to the destination host (by using the JavaRMI mechanism in our implementation). In addition to the stack frames, the receiver objects executing the migrating methods are also transmitted. Object `o2` executes on both the departure and destination hosts.

3.3 Declaring Method by Using Migratory Primitive

When migration takes place in the extent of a method, the method itself must be declared as such. This is done using the *migratory* primitive. That is, the go statement and the invocation of a migratory method must be written in a migratory method. If they are put in a non-migratory method, they must be enclosed within an undock statement.

3.4 Creating Objects with Specifying Data Movement Protocols

A data movement protocol is assigned to an object when it is created as an extra parameter of the new expression.

```
x = new(copy) X;
y = new(resident) Y[3];
z = new(carry) Z*;
```

In the first line of the above code, an area for class X is allocated and the pointer of the copy protocol to the area is assigned to variable `x`. In the second line, an area for an array of class Y is allocated and the pointer of the resident protocol to the area is assigned to variable `y`. The whole area of the array shares the same data movement protocol because it is difficult to manage a partially-available array, an array some of whose elements are available and the other elements are unavailable. Consider the case where an array of the copy protocol and some of the elements are the carry protocol. When the array is transmitted on migration, the whole array is copied and only the pointers referring to the elements of the carry protocol are forwarded to the destination. But the elements of the copy protocol can be computed from the base address of the array and the indices. This can cause an unexpected effect on the programmer. In the last line, an area for a pointer of class Z is allocated and the pointer of the carry protocol to the area is assigned to variable `Z`.

The data movement protocol assigned to a pointer cannot be changed after it is assigned at the time of initialization. The reason is threefold: (1) as mentioned

above, it is difficult to maintain a partially-available array. (2) The change of data movement protocol interferes with the implementation of data movement protocol. For instance, let us consider the following implementation of data movement protocol. An address space is splitted into six ones, each of which represents the area for a specific data movement protocol. A value of a certain data movement protocol is stored into the splitted address space assigned with the data movement protocol. In this implementation, it is easy to check a pointer belongs to which data movement protocol. Changing the data movement protocol of a value, however, requires the cost of garbage collection because the value is moved to a different address space and all the references to the value must be forwarded. (3) In the existing mobile language systems, the construct equivalent to the data movement protocol cannot be changed by the programmer.

Chapter 4

An Implementation of the Programming Language

In this chapter, an implementation of the mobile programming language based on our mobile calculus is described. It is realized as a translator which takes as input code a Java program written in a Java language extended with language constructs for migration, and outputs pure Java source code that uses JavaRMI. The translator itself is written in Standard ML. Our scheme enables a Java program to be migrated across computers while preserving its execution state, including the values of local variables and the dynamic extents of try-and-catch blocks. This scheme provides the programmer with flexible control of migration, including transparent migration. It is based on source-code-level transformation. The translated code can run on any Java interpreter and can be compiled by any just-in-time compiler. We have measured some execution performance for several application programs, and found that the translated programs are only about 20% slower than the original programs. Because migration is completely controlled by using only three language constructs added to the Java language (*go*, *undock* and *migratory*), the programmer can write programs to be migrated easily and succinctly. Our system is available in the public domain. The use of data movement protocol is restricted on the copy and the resident protocols because of the restriction of JavaRMI. To enable the full protocols, the JavaRMI mechanism must be extended.

4.1 Background and Objectives

In mobile agent systems, communication is based on *migration*. Migration is called *transparent* [45] if a migrated program resumes its execution at a destination site with exactly the same execution state as that before migration began. Compared to non-transparent migration, transparent migration is more desirable [32, 40, 45] for writing programs to be migrated and understanding the semantics of migration. The migration transparency of existing mobile systems is, however, not satisfactory.

- Almost all existing mobile systems [64, 76, 92] based on Java do not preserve the calling stack.
- Telescript [100, 101], an early mobile agent system, provides transparent migration, but the unit of migration is restricted to only one object (agent) [41] at a time.

- In the few systems [32, 40, 45, 87] that support transparent migration, the continuation (the complete execution state of the remaining computation) always migrates to the destination, which incurs considerable performance loss, as pointed out by Cejtin et al. [21]

We have extended the Java language by adding three constructs for controlling migration. They are based on our mobile calculus [84] and on mobile languages, such as that developed by Watanabe [99], and allow more flexible control of migration, including transparent migration. We call this flexible migration mechanism *controllable transparent migration.*

With Java, it is difficult to implement a transparent migration mechanism without degrading portability or efficiency for several reasons.

- Consider the case where the runtime system is extended in order to support migration. In this case, we can expect that the execution performance of a migrating program is almost the same as before, and that existing class files can be used for the extended system without recompilation. In return, however, migrating programs are not portable because they can only be executed on a specific interpreter or on the runtime system of a specific just-in-time compiler. This scheme thus does not fit the philosophy of Java, that is, "write once, run anywhere".

- When a migration mechanism is provided as a class library, it offers nearly the same benefits as when a runtime system is extended. Unfortunately, there are several obstacles that are difficult to overcome in this approach. Transparent migration requires transmission of the stack. Dynamic inspection of the stack by Java bytecode itself is forbidden by the *Java security policy.* [1] Therefore, it is difficult to implement transparent migration as a class library.

- In an approach based on source-code-level translation (e.g., Arachne [32]), a Java program is transformed in such a way that the translated Java program can explicitly manage its execution states as Java objects, which enables transparent migration. The major drawback is a slowdown in execution speed due to the code fragments inserted to maintain the information for migration. In a straightforward implementation [32], execution of a transformed program was about twice as slow as that of the original program.

One of the major drawbacks of not providing transparent migration is that it is difficult to migrate existing non-mobile application programs. When migration transparency is realized partially, the execution environment after migration is not as the same as before. Therefore application programs can encounter unexpected situations. If the programmer want to migrate existing, non-mobile applications, they must be rewritten so that they work correctly even after migration. The work is not simple in general.

Our migration scheme is based on a source-code-level translation that offers the following features:

portability Programs translated using our scheme require only standard Java and JavaRMI, so a translated program can be run on any Java interpreter and can be compiled by any just-in-time compiler.

[1] A native method can inspect the execution stack at runtime, but the memory layout of objects in the stack heavily depends on the implementation of the runtime system.

efficiency In our scheme, most of the execution state is saved at the time of migration by using an exception-handling mechanism. Therefore, the overhead for ordinary execution is low (about 20% in most cases).

accuracy In migration schemes based on source-to-source transformation, it is difficult to simulate exception handling (try-and-catch and throw statements) appropriately. In our scheme, the dynamic extents of the try-and-catch blocks are completely preserved at the time of migration.

Testing of our scheme showed that it can migrate Java programs with a graphical user interface (like migratory applications [12, 13] and XMOVE [102]) between IBM-PC (Pentium / Windows-NT) and Sun workstation (SPARC / Solaris).

The rest of this chapter is organized as follows. In Section 4.2 we briefly overview our migration scheme. In the subsequent three sections, we describe our program transformation for transparent migration in detail. In Section 4.8 we discuss the difficulties induced by the source-code-level transformation, and in Section 4.9 we present experimental results showing the execution performance and growth in code size. In Section 4.10, we present photographs in which Java applications are migrated. In Section 4.11, we compare it with a related one. We also briefly discuss the need for extending JavaRMI.

4.2 Overview of Our Transparent Migration

To achieve transparent migration, a method being executed must be suspended, transmitted, and then resumed. In the next three sections we describe our migration scheme based on source-level transformation. In this section we give a brief overview.

The unit of our source-to-source translation is a *method*. A method is transformed in such a way that the translated method can explicitly manage its execution state as Java objects. The migratory modifier of a method is used to determine if the method is to be transformed for migration. Migratory declarations make source-level translation easier. The system proposed by Fünfrocken [40] is also based on source-to-source translation, but it requires the fixed-point iteration of a call-graph to determine the set of methods that should be translated because it lacks this kind of declaration.

4.2.1 Saving the Execution State

The execution state of each method consists of the execution point from which the method resumes and the values of all local variables. A method is transformed in such a way that (1) all the variable declarations are elevated and moved to the head of the method and (2) it captures `NotifyMigration` exceptions. The current execution point is continuously saved to a newly introduced special variable during ordinary execution of the method. When migration takes place, an exception is raised and captured by the method. This causes the method to store the values of all local variables into a state object. The method then propagates the exception to the caller. A state object is defined for each method, and all the state objects created during the state-saving process are connected as a chain. The details of this state-saving will be described in Section 4.4.

4.2.2 Transmitting the Execution State

Eventually, the exception is either captured by an undock statement or it reaches the bottom of the stack. Then, the chained state objects are serialized and transmitted to the destination host by using JavaRMI. The receiver objects executing the migrating methods are also transmitted.

4.2.3 Restoring the Execution State

The execution state is recovered from the chain of state objects. A method is transformed so that (1) it takes the corresponding state object as an extra parameter and (2) it can resume its execution from any statement in its body, as will be described in Section 4.3. When a method m was being suspended by calling another method m' at the time of migration that was caused by calling m', a code fragment is inserted so that m' is called before resuming the execution of m. The value of each local variable is properly restored before executing the body of the method, as described in Section 4.4. The state of the execution stack (and the dynamic extents of the try-and-catch statements) is reconstructed at the destination host by calling the method at the bottom of the stack with the corresponding state object as the extra argument. This reconstruction is described in Section 4.5.

4.3 Resuming a Method

To enable a migrated method to resume execution, a mechanism that enables execution to jump to any program point is needed. In other words, a way to resume *control* is needed.

Migration in the C language by using source-code translation [32] exploits a simple jump mechanism: goto statements. Java, however, does not have goto statements. Its break and continue statements permit only escaping from a block statement, there is no way to jump into a compound statement. Our scheme implements the jump facility with low overhead by using switch-case statements and the unfolding technique. This facility is based on transforming a method into a form in which the method can be resumed from any program point. Doing this requires two preprocessings.

4.3.1 Preprocessings

Splitting an Expression with Side-Effects.

Resuming execution of a statement needs special care that contains side effects. Consider the following assignment statement (where `foo` is a migratory method):

```
y = a[x++] + foo(x) + b[x++];
```

If the method containing this statement is to be resumed immediately after method `foo` is invoked, we must know the values of `a[x++] + foo(x)` before migration and assigns the sum and `b[x++]` to variable `y` after migration. Although these intermediate values do not appear as local variables in the method, they must be captured and restored after migration. To handle these intermediate values, a statement including side effects is decomposed into a sequence of atomic operations. The above statement is decomposed as follows.

```
tmp1 = a[x++];
tmp2 = foo(x);
tmp3 = b[x++];
y = tmp1 + tmp2 + tmp3;
```

New variables (`tmp1`,...,`tmp3`) are generated to keep track of the intermediate values. This transformation guarantees that we can avoid resumption from within an expression. This does not seem to be the case in Fünfrocken's scheme [40].

Elevating Variable Declarations.

Next, we identify all the local variables (including the intermediate variables introduced in the first preprocessing) in the method, elevate the variable declarations, and move them to the head of the method. Unlike the C language, Java allows variable declarations to be placed almost anywhere in a method. Because the statements in a method may be reordered or duplicated during the subsequent transformations, elevating the variable declarations avoids the difficulties of code reordering and duplication. If we duplicated a block of statements including a variable declaration, the resulting code fragment would contain two declarations for the same variable. Elevating avoids this. When a variable is declared with its initial value, we elevate only its declaration and leave the assignment of the initial value at the original program position.

4.3.2 Jump Facility

Our scheme enables execution to jump to any top-level statement in the method. A top-level statement is one that is not enclosed in an other statement, such as a if statement or a for statement. Note that the goto facility of the C language can be simulated in Java, but only at the top-level. Consider the following Java code fragment.

```
TopLevel: for( ;; ) switch (EntryPoint) {
    case 0:
        ...
    case 1:
        ...
    case 2:
        ...
}
```

Each case statement can be considered a label used as a destination of a goto statement. An occurrence of `goto` n`;` is encoded as {`EntryPoint =` n`; continue TopLevel;`}. By following this approach, a method in which every statement appears at the top level can be transformed into a resumable form. Consider the following method.

```
void bar() {
    foo(); // migratory method invocation
    System.out.println("after foo");
}
```

To resume this method after invoking `foo`, we make the following transformation by adding a switch statement, assuming variable `EntryPoint` is set to 1.

```
void bar() {
 TopLevel: for( ;; ) switch (EntryPoint) {
    case 0:
        foo(); // migratory method invocation
    case 1:
        System.out.println("after foo");
        return;
    }
}
```

In our migration scheme, a method does not have to be resumable from every statement because a method is resumed only immediately after a migratory method is invoked. The reason for this will be explained in Section 4.5.

4.3.3 Unfolding Technique

To enable execution to be resumed in the middle of a compound statement, such as if statements and for statements, (1) the sub-statements after the resumption point and (2) the subsequent statements that should be executed until control reaches the top level are duplicated at the top level. By representing a compound statement as a while construct, we illustrate the necessary transformation below. Consider the following code fragment.

```
while(C1) {
    while(C2) {
        A;
        foo(); // migratory method invocation;
        B;
    }
}
return;
```

`A` and `B` are arbitrary non-migratory statements, and `C1` and `C2` are arbitrary non-migratory expressions. By unfolding the loops, we translate this code fragment into the following code fragment.

```
label1:
   while(C1) {
       while(C2) {
           A;
           foo(); // migratory method invocation;
           B;
       }
   }
   return;
label2:
   while(C2) {
       A;
       foo(); // migratory method invocation;
       B;
   }
   goto label1;
label3:
   B;
   goto label2;
```

(This unfolding duplicates the contents of the inner loop at the top level.) The resumption point is at `label3` when method `foo` is called at the time of migration. All the labels are at the top level so they can be implemented by top-level jumps. By jumping to `label3`, execution is apparently resumed immediately after `foo()`. As illustrated above, if we have top-level jumps, a method is resumable from any point by using unfolding.

The size of a transformed method increases in proportion to $O(n^2)$, where n is the maximum depth of the loops, because the body of a loop with a depth of n is unfolded n times.

Because labels must be at the top level, a try block cannot bridge several labels. If a loop appears in a try block, the block is duplicated so that it does not bridge labels. In our current implementation, to reduce the size of a transformed method, an optimization is performed such that the bodies of the catch blocks are shared among the duplicated try-and-catch statements.

4.4 Saving and Restoring Local Variable Values

Saving and restoring the values of local variables in a method play an important role in implementing migration. In this section, we first touch upon a related study in which migration is implemented by source-code translation and then describe our scheme.

4.4.1 The Arachne Scheme

In the Arachne scheme [32], migration in the C language is based on source-code translation. A native stack frame is not used to save the values of local variables; instead a stack is managed at the user-program level. A special object representing a stack frame is allocated for every function invocation. Each access to a local variable is replaced with the corresponding field access to the special object. A stack is transmitted to a remote site by transmitting these special objects. Unfortunately, applying the Arachne migration scheme to Java causes a considerable performance loss, as discussed in Section 4.9.

4.4.2 Our Scheme

Our scheme for saving the values of local variables is as follows. The body of each migratory method is enclosed by a try statement to capture a special exception, `NotifyMigration`, that is signaled by the occurrence of migration. When the method actually captures the exception, the values of all local variables are stored into a new state object, and the exception is raised again. This procedure repeats until the exception reaches the bottom of the stack or is captured by an undock statement.

For each method, a class for the state object is defined. The instance variables of the class record (1) the values of all the local variables, including `EntryPoint`, (2) a reference to the receiver object of the method (i.e., the value of `this`), and (3) the state object of the sub-method invoked by this method on resumption. Every state class inherits a common base class, so that state objects can constitute a chain.

When restoring the values of local variables, a state object containing the values is applied to the method as an extra argument. The values of the local variables are

set to those stored in the object at the head of the method. For ordinary method invocation, the null value is passed to the extra argument.

Putting saving and restoring together, the transformation of a method looks like the following.

```
void foo() {                        void foo(State_X_foo State) {
    int x = 0;                          int x;
    ... the body of a method ...        if ( State == null )
}                                           x = 0;
                                        else
                                  ⇒         x = State.x;
                                        try {
                                            ... the body of a method ...
                                        } catch (NotifyMigration e) {
                                            State = new State_X_foo(this);
                                            State.x = x;
                                            e.Append(State);
                                            throw e;
                                        }
                                    }
```

In the code on the righthand side, `State_X_foo` is the name of a state object. A state object is defined for each method and contains all the local variables of that method. Exception `e` maintains the chain of state objects. A state object is appended to the chain by executing `e.Append(State)`. The chain of state objects is transmitted to a destination site by using the JavaRMI mechanism. At the destination, the execution state is reconstructed based on the chain of state objects, and execution resumes there.

4.5 Reconstructing Stack of Method Invocations

A method is resumed by using a combination of the techniques described in the previous two sections. Resuming a *call stack* of method invocations, however, requires a different technique because the dynamic extents of try-and-catch blocks spanning method invocations must be reconstructed. If these dynamic extents are not preserved, the semantics of throwing an exception is violated. The call stack of method invocations is reconstructed by calling each method in the stack with its own state object as the extra argument in the order that the methods were invoked before migration. Suppose method m' was called by method m, and migration takes place. We need to insert a code fragment that executes the rest of m' at the resumption point in method m. When method m is called with its state object, m resumes m' and after returning from m', the rest of m is executed. To implement such a behavior, method `bar` from Section 4.3.2 is translated as follows.

```
void bar(State_X_bar State) {
 TopLevel: for( ;; ) switch (EntryPoint) {
    case 0:
        EntryPoint = 1;
        OldState = State;
        State = null;
    case 1:
        foo ((State_X_foo)OldState.Child); // migratory method invocation
        System.out.println("after foo");
        return;
```

```
        }
    }
```

`State_X_bar` and `State_X_foo` are the names of the state classes defined for methods `bar` and `foo`, respectively. Suppose that a migration takes place when method `foo` is being called. On restart, a state object is passed to the corresponding method `bar` (the state object is referred to by variable `State` in the above code fragment). The state object shows that the value of `EntryPoint` has been set to 1. Therefore, control is transferred to case 1. Then, method `foo` is invoked with `State.Child`, where `State.Child` refers to the state object of the callee of this method, namely the state object of method `foo`.

4.6 Translating Other Program Constructs

This section describes the translation scheme of other program constructs that need special care.

4.6.1 Translating an Undock Statement

An undock statement is translated to a try-and-catch statement whose catch clause captures exception NotifyMigration. When it captures NotifyMigration, the execution state is encoded in the chain of state objects. The catch clause tries to transmit the chain to the destination computer by using JavaRMI. If suceeds, the control exits the catch clause. It looks as if the control is escaped from the undock statement for the programmer. If it fails to transmit the chain of state objects, an exception designating the reason of the failure is thrown. The catch caluse also captures the exception and resumes the encoded execution state at the *current* host. After the stack is reconstructed and before the execution is resumed, the exception is thrown as if it is thrown from the go statement.

4.6.2 Translating a Synchronized Statement

There is a difficulty in translating a synchronized statement. The contents of a synchronized statement must be executed *atomically* for the other threads. It is thus difficult to simulate the contents of a synchronized statement by composing the unfolding technique and the top level jump because the contents of a synchronized statement cannot be divided.

Our translation scheme of a synchronized statement is as follows. A new method is added to the class for each synchronized statement. The contents of the new method is the synchronized statement itself and the original synchronized statement is replaced with an invocation of the additional method. The new method is transformed in the same way as the case of normal methods except that the contents of the transformed method is enclosed within a synchronized statement.

4.6.3 Translating a finally clause

The finally clause in a try-catch-finally statement is a big trouble in implementing and verifying [90] the Java language. The contents of the finally clause must be executed even when the corresponding try block terminates the execution abruptly. It also requires a special care for our transparent migration scheme. Therefore, we

avoid to handle finally clauses directly, but a method including finally clauses is translated to an equivalent code fragment without finally caluses. Our translation scheme is appilied to the code fragment without finally clauses.

4.7 Migration Server

Surprisingly, the structure of the migration server is quite simple. We quote the important part of the source code.

```
public static void Transmit(StackFrame BottomofStack,String Destination) {
    try {
        Executor        exec = (Executor)Naming.lookup (Destination);
        exec.Execute (BottomofStack);
        System.out.println("migrated successfully");
    } catch ( Exception e ) {
        System.out.println ("migration failure: " + e.getMessage());
    }
}
```

The above method is all the code that transmits the stack frames to the remote computer. The chain of the stack frames bound to `BottomofStack` is just passed to a remote method `Execute` in interface `Executor`. The class files assigned to the stack frames are automatically transmitted by the mechanism of JavaRMI.

On the other hand, the method `Execute` is as follows.

```
public void Execute(StackFrame BottomofStack) throws RemoteException {
    BottomofStack.Start();
}
```

provided that method `Start` invokes the suspended method assigned to the stack frame. The method simply invokes the suspended method at the bottom of the stack. By invoking the suspended method assigned to the *bottom* of the stack, the execution state is reconstructed as explained.

4.8 Limitations due to Source-Level Translation

There are limitations in the current implementation of our migration scheme. Most of the limitations are essentially due to source-level translation. Therefore, similar limitations apply to other migration schemes based on source-level translation.

There are three areas in a program from which a go statement and a migratory method cannot be called, meaning that migration cannot take place.

- in a class initializer (a static initializer)
- in an instance initializer, and
- in a constructor

It is difficult to resume in a class or instance initializer because it is difficult for a user program to reconstruct the effects of executing these initializers. They are invoked by the runtime system when a class is loaded or an object is created. A constructor is also invoked by the runtime system, so it cannot be resumed. To allow invoking a migratory method from these areas, we must simulate the effects of these initializers and a constructor by combining other language constructs.

program	elapsed time(ms)			byte code size(bytes)		
	original	transformed	growth	original	transformed	growth
fib(25)[1]	579	1170	+102%	1274	2414	1.89
qsort(200000)[1]	8207	8727	+6.3%	2765	3726	1.35
nqueen(11)[1]	20854	24522	+17.0%	1647	2925	1.77
Richards[2]	4777	4783	+0.125%	10868	25482	2.34
DeltaBlue[2]	25881	28047	+8.37%	28257	50983	1.80

(1: JDK 1.2, Sun UltraEnterprise, 2: JDK 1.1.7a, AMD-k6 200MHz)

Table 4.1: Execution performance and growth in byte code size of programs transformed by our scheme.

Another limitation is that locking cannot be preserved on migration. If a lock has been acquired by a synchronized statement or a synchronized method, it is released at the time of migration by the exception notifying migration. Although the locked state of an object is correctly recovered after migration, the state of the threads that were waiting on the migrating object cannot be preserved because the exception used in our migration scheme releases the object lock. The locked state is thus temporarily lost on migration.

This lock problem, however, is avoidable even in source-level translation scheme. The lock in question is the most primitive, Java language-level lock. Java language does not provide a lock spanning among several hosts in the language level. Such a distributed lock must be composed from the other language constructs. Thus, if the programmer needs a distributed lock, it is not the most primitive lock.

4.9 Performance

To evaluate the overheads imposed by our translation scheme, we carried out benchmark testing by executing several Java application programs. The execution performance of programs transformed by our scheme was compared with that of the original programs. Migration does *not* take place during the execution of the benchmark programs. The results are shown in Table 4.1. The overheads with the Fibonacci method is rather high because the body of Fibonacci is very small. Most of the overhead is due to extra control transfers induced by the unfolding. When the body of a method is very small, the overhead of code insertion is tend to be high. For the quick sort and N-queen applications, the elapsed times were approximately 20% higher than those of the original applications. Richards is a medium-scale benchmark program simulating the task dispatcher in an operating system. DeltaBlue [39] is a medium-scale constraint solver benchmark program (about 1000 lines in the case of Java). Java versions of Richards and DeltaBlue are available with the source code from Sun Microsystems Laboratories [63]. (Richards has seven variants. We used the one called "richards_gibbons".)

The growth in code size due to program transformation is also shown in Table 4.1. The growth rate was less than twice except for the Fibonacci method and Richards, but even in these cases it was less than 3 times. In general, the code size grows in proportion to the square of the depth of the loops.

For comparison, we applied the Arachne scheme to Java and measured the ex-

	elapsed time(ms)
original quicksort	7691
Arachne-style quicksort	16300
cost of field selection	5825
cost of state object allocation	2784

(JDK 1.1.7a, AMD-k6 200MHz)

Table 4.2: Performance of Arachne scheme.

ecution performance. As shown in Table 4.2, the direct adoption of the Arachne scheme to Java causes a considerable performance loss. The elapsed time for executing a method transformed by using the Arachne scheme was about twice that of the original method. The overheads can be roughly divided into two parts. The first and major overhead is the cost of field selection. The second is the cost of allocating state objects at the head of the method.

4.10 Sample Mobile Applications

A sequence of demo snapshots in which three applications are migrating between two computers is presented in Figure 4.1. The applications are the graph layout program included in JDK as a demo application, the tower of Hanoi, and quick sorting. The latter two applications are written using conventional recursive functions. We used two hosts, and the applications migrated from one to the other at different times. Each application had a window (an instance of java.awt.Frame class). Because a window is serializable, it can be transmitted to a remote host by using JavaRMI. When a visible window is transmitted to a remote host by using JavaRMI, the window still remains at the departure host and the moved window is not mapped on the display of the destination host automatically. Therefore, when an application migrates, the window remaining at the departure host must be explicitly deleted, and the window at the destination host must be explicitly mapped. This behavior is simply implemented by using `go` and `undock`. The code fragment that deletes the window at the departure host is as follows.

```
undock {
    ... the body of an application ...
}
dispose();
```

When an application migrates, the control at the departure host exits the undock statement shown above. The code fragment that maps the window at the destination host is as follows.

```
go (destination);
show();
```

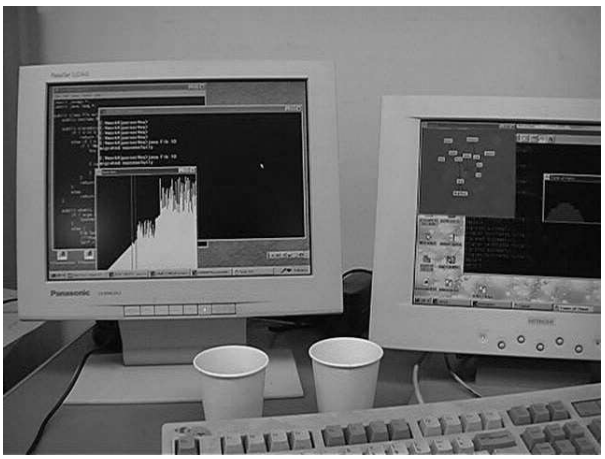
If migration succeeds, `show()` is executed at the destination.

In the picture (a) of Figure 4.1, all the applications are running on the left host whose monitor is shown on the left. In the picture (b), the graph layout program has migrated to the right host and the other applications remain on the left host. In the picture (c), Hanoi has also migrated to the right host. Finally, the quick sorting has migrated to the right host and the graph layout has came back to the left host in the picture (d).

(a)

(b)

(c)



(d)

Figure 4.1: Applications migrating between two computers.

4.11 Related work and a need of extending JavaRMI for further development

Although our scheme and Fünfrocken's [40] were developed independently, both are based on Java source-code translation and use exception handling to transmit the execution stack. An important difference between them is in the way that method execution is resumed. In Fünfrocken's scheme, a so-called *artificial* program counter is used to skip the code fragments that have been already executed. An additional code fragment that checks whether the current statement should be skipped is inserted for each statement. (Successive statements can be grouped if they do not include migratory method invocations.) Our scheme does not need such checking, but in some cases, a compound statement needs to be unfolded. In saving a state, Fünfrocken's scheme inserts additional code that records the values of local variables into the state object for each migratory *method invocation*. In contrast, additional code is inserted for each *method* in our scheme because all variable declarations are moved to the head of the method. Fünfrocken's scheme performs fixed-point iteration of a call-graph to find the set of migratory methods. Our scheme does not need such an iteration because a migratory method is declared as such. The difference in performance cannot be directly compared because Fünfrocken did not report execution performance. He did report the growth in code size due to source-code transformation. The growth factor was about 3.65 to 4.7 times compared to the original programs. In our scheme, the growth factor is about 1.45 to 2.8 times.

From our experience in writing mobile applications with graphical user interfaces, we feel that the JavaRMI mechanism needs to be extended for transmitting graphical user interfaces. When a visible window is transmitted to a remote host by using JavaRMI, the window remains at the departure host and the moved window is not mapped on the display of the destination host automatically. Therefore, when an application migrates, the window remaining at the departure host is explicitly deleted, and the window at the destination host is explicitly mapped. The applications we wrote had only one window; if applications with several windows were written, this code insertion would be cumbersome. Therefore, we need a mechanism that specified methods are automatically invoked before and after migration, analogous to that of the :before and :after methods in the common lisp object system (CLOS) [58]. Such a mechanism would be useful for any resource that needs initialization and finalization, not for only graphical user interfaces.

4.12 The Translated Fibonacci Function

The final transformed program for the mobile Fibonacci method in Section 3.1 is given below. It consists of the transformed method and the class definition for a state object of method `fib`. They are automatically generated by our translator written in Standard ML. There are several minor differences between the scheme described in this chapter and the actual implementation of the translator. We believe, however that the differences would be easily understood for those who are familiar with Java.

```
public static int       fib(StackFrame M_ParentState,int n) throws Exception,
NotifyGone {
    STFibfibi   M_State = null;
    STFibfibi   M_OldState = null;
    int         M_EntryPoint = 0;
```

```
int         tmp1 = 0;
int         tmp2 = 0;
if (M_ParentState == null) {
    M_State = null;
    M_EntryPoint = 0;
}
else { // restoring local state
    M_State = (STFibfibi)M_ParentState.Child;
    tmp2 = M_State.tmp2;
    tmp1 = M_State.tmp1;
    n = M_State.n;
    M_EntryPoint = M_State.M_EntryPoint;
}
try {
 M_TopLevel:
    for( ;; ) switch (M_EntryPoint) {
    case 0:
    case 1:
        if (n == 0) {
            if (!Moved) {
                M_OldState = M_State;
                M_State = null;
                M_EntryPoint = 23;
                StackFrame.Migrate(M_OldState,"//flute:2001/SimpleServer");
                M_OldState = M_State;
                M_State = null;
                StackFrame.MigrationStatus(M_OldState);
                Moved = true;
            }
            return 1;
        }
        else if (n == 1) return 1;
        else {
            M_OldState = M_State;
            M_State = null;
            M_EntryPoint = 14;
            tmp1 = fib(M_State,n - 1);
            M_OldState = M_State;
            M_State = null;
            M_EntryPoint = 18;
            tmp2 = fib(M_State,n - 2);
            return tmp1 + tmp2;
        }
    case 2:
        return 0;
    case 3:
    case 4:
    case 5:
    case 6:
        return 1;
    case 7:
    case 8:
    case 9:
    case 10:
    case 11:
```

```
        case 12:
        case 13:
        case 14: // resumption point
            tmp1 = fib(M_State,n - 1);
            M_OldState = M_State;
            M_State = null;
            M_EntryPoint = 18;
        case 18: // resumption point
            tmp2 = fib(M_State,n - 2);
            return tmp1 + tmp2;
        case 20:
        case 21:
        case 22:
        case 23: // resumption point
            M_OldState = M_State;
            M_State = null;
            StackFrame.MigrationStatus(M_OldState);
            Moved = true;
            M_EntryPoint = 6;
            continue M_TopLevel;
        }
    } catch (NotifyGone M_Gone) { // saving local state
        M_State = new STFibfibi();
        M_State.tmp2 = tmp2;
        M_State.tmp1 = tmp1;
        M_State.n = n;
        M_State.M_EntryPoint = M_EntryPoint;
        M_Gone.Append(M_State)
        throw M_Gone;
    }
}

public class STFibfibi extends javago.StackFrame {
    public      STFibfibi() {
    }
    public Object       Continue() throws Exception,javago.NotifyGone {
        StackFrame      Frame = new StackFrame();
        Frame.Child = this;
        return new Integer(Fib.fib(Frame,0));
    }
    public int  M_EntryPoint;
    public int  n;
    public int  tmp1;
    public int  tmp2;
}
```

Chapter 5

Conclusion

The thesis is summarized as follows.

- We have developed a model of mobile computation where dynamic transmission of code, data and execution state can be described. The distinctive feature of our calculus is its capability to represent control and stores. In the control aspect, we have proposed

  - an operator that flexibly specifies the area of execution state to be transmitted,
  - an implementation scheme for the mobile language by using CPS, and
  - a novel optimization technique that reduces the size of the transmitted code on migration.

  It is difficult to carry out such analyses in the mobile calculi based on concurrent calculi [19, 82, 85] because the notion of control is ambiguous in concurrent calculi as pointed out in Refs. [52, 71, 79].

  In the data aspect, we have proposed

  - the notion of *data movement protocols* that describes *any* transparent data migration.

  The data movement protocols specify how a value in a heap is transmitted to a remote computer on migration. The notion of a heap does not appear in many mobile calculi although it may be encoded in them. Our work investigates the very encoding, the system with heaps.

- A mobile language based on the above results is designed. The programmer can describe migration succinctly because migration is completely controlled only by a few annotations. Nevertheless the expressiveness of the mobile language is high because *any* transparent migration can be described in the language and also the transparency of migration is controllable.

- The mobile language is actually implemented as an extension of the Java language. We believe that the implementation was the first one for Java that enables transmission of the execution stack. Moreover, the overhead on program execution imposed by our implementation scheme is low. The compiler takes the extended Java language and outputs the standard Java program that

uses Java RMI. The translator is written in Standard ML and is available in the public domain [93].

Nowadays, distributed computation seems to be popular, but it is still far from those described in science fictions such as *Neuromancer* by William Gibson and *Tron* by Steven Lisberger. (They should not be received at the level of their concrete appearance such as the *unrealistic* human-machine, neural-machine interface, but at the level of what is realized by it. For instance, one can intrude the other's *environment* and see as if he sees, namely, in a transparent way. Moreover, one can manipulate the world as if the intruded person does.) Programming languages, stable implementations and useful applications for mobile computation are necessary. Distributed computation still leaves an immense room for further investigation.

Bibliography

[1] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 111–130, 1997.

[2] Anurag Acharya and Joel Saltz. Dynamic Linking for Mobile Programs. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 245–262, 1996.

[3] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and Language-Independent Mobile Programs. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 127–136, 1996.

[4] Roberto M. Amadio. Translating Core Facile. Technical report, European Computer-Industry Research Centre, 1994. ECRC-94-3.

[5] Roberto M. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 374–391, 1997.

[6] Roberto M. Amadio and Sanjiva Prasad. Localities and Failures. In *Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216, 1994.

[7] Roberto M. Amadio and Sanjiva Prasad. Localities and Failures. Technical report, European Computer-Industry Research Centre, 1994. ECRC-94-18.

[8] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[9] Joseph Bates. The Role of Emotion in Believable Agents. Technical report, School of Computer Science, Carnegie Mellon University, 1994. CMU-CS-94-136.

[10] Russell Beale and Andrew Wood. Agent-Based Interaction. In *People and Computers IX: Proceedings of HCI'94*, pages 239–245, 1994.

[11] Dave Berry, Robin Milner, and David N. Turner. A Semantics for ML Concurrency Primitives. In *Proceedings of the 19th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 119–129, 1992.

[12] Krishna Bharat and Luca Cardelli. Migratory Applications. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 131–148, 1997.

[13] Krishna A. Bharat and Luca Cardelli. Migratory Applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, 1995. Digital Systems Research Center Research Report 138.

[14] Ramón Cáceres and Venkata N. Padmanabhan. Fast and Scalable Handoffs for Wireless Internetworks. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 56–66, 1996.

[15] Ting Cai, Peter A. Gloor, and Saurab Nog. DartFlow: A Workflow Management System on the Web using Transportable Agents. Technical report, Dartmouth College, 1996. PCS-TR96-283.

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

[17] Luca Cardelli. Mobile Computation. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 3–6, 1997.

[18] Luca Cardelli. Program Fragments, Linking, and Modularization. In *Conference Record of POPL'97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, 1997.

[19] Luca Cardelli and Andrew D. Gordon. Mobile Ambients, 1997.

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

[21] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-Order Distributed Objects. In *ACM Transactions on Programming Languages and Systems*, volume 17(5), pages 704–739, 1995.

[22] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–47, 1996.

[23] Colusa. Omniware: A Universal Substrate for Mobile Code, 1995. Colusa Software white paper.

[24] Bruno Courcelle. Fundamental Properties of Infinite Trees. In *Theoretical Computer Science*, volume 25, pages 95–169, 1983.

[25] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing Mobile Code Languages. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 93–109, 1996.

[26] George Cybenko, Robert Gray, Yunxin Wu, and Alexy Khrabrov. Information Architecture and Agents. Technical report, Dartmouth College, 1994.

[27] Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192(2), 1998.

[28] Olivier Danvy and Andrzej Filinski. A Functional Abstraction of Typed Contexts. Technical report, Institute of Datalogy, University of Copenhagen, 1989. DIKU 89/12.

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

[30] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, 1997.

[31] Mikael Degermark, Mathias Engan, Björn Nordgren, and Stephen Pink. Low-loss TCP/IP Header Compression for Wireless Networks. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 1–14, 1996.

[32] Bozhidar Dimitrov and Vernon Rego. Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. In *Proceedings of IEEE Parallel and Distributed Systems*, volume 9(5), pages 459–469, 1998.

[33] David Eichmann. Ethical Web Agents. In *Proceedings of the Second International World-Wide Web Conference*, 1994.

[34] Matthias Felleisen. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[35] 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.

[36] 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.

[37] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL'96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[38] Armando Fox and Steven D. Gribble. Security on the Move: Indirect Authentication Using Kerberos. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 155–164, 1996.

[39] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *CACM*, 33(1):54–63, 1990.

[40] Stefan Fünfrocken. Transparent Migration of Java-Based Mobile Agents. In *MA'98 Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37, 1998.

[41] General Magic Inc. *Telescript Programming Guide*, 1996. Version 1.0 alpha 2.

[42] Carlo Ghezzi and Giovanni Vigna. Mobile Code Paradigms and Technologies: A Case Study. In *Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 39–49, 1997.

[43] Richard Goodwin. Formalizing Properties of Agents. Technical report, Carnegie Mellon University, 1993. CMU-CS-93-159.

[44] James Gosling and Henry McGilton. The Java Language Environment, 1995.

[45] Robert S. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, 1995.

[46] David Alan Halls. *Applying Mobile Code to Distributed Systems.* PhD thesis, Computer Laboratory, University of Cambridge, 1997.

[47] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea?, 1994. IBM Corporation white paper.

[48] Masatomo Hashimoto and Atsushi Ohori. A Typed Context Calculus. Available from Research Institute for Mathematical Sciences, Kyoto University, as a preprint no. RIMS-1098., 1998.

[49] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[50] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A Dynamic Disk Spin-down Technique for Mobile Computing. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 130–142, 1996.

[51] Matthew Hennessy and James Riely. Resource Access Control in Systems of Mobile Agents. Technical Report Report 2/98, University of Sussex, 1998.

[52] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *ACM SIGPLAN Notices*, volume 25(3), pages 128–136, 1990. Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP.

[53] Barron C. Housel and David B. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 108–116, 1996.

[54] Leon Hurst, Pádraig Cunningham, and Fergal Somers. Mobile Agents - Smart Messages. In *Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 111–122, 1997.

[55] Suresh Jagannathan. Communicating-Passing Style for Coordination Languages. In *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 131–149, 1997.

[56] Gregory F. Johnson. GL – A Denotational Testbed with Continuations and Partial Continuations. In *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 165–176, 1987.

[57] Anthony D. Joseph and M. Frans Kaashoek. Building Reliable Mobile-Aware Applications using the Rover Toolkit. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 117–129, 1996.

[58] Guy Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[59] Kazuhiko Kato. Safe and Secure Execution Mechanisms for Mobile Objects. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 201–211, 1996.

[60] R. Kelsey and P. Hudak. Realistic Compilation by Program Transformation. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 281–292, 1989.

[61] Frederick Knabe. Performance-Oriented Implementation Strategies for a Mobile Agent Languages. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 229–244, 1997.

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

[63] Sun Microsystems Laboratories. Benchmarking Java with Richards and Deltablue, 1997.

[64] Danny B. Lange and Daniel T. Chang. IBM Aglets Workbench: A White Paper, 1996. IBM Corporation.

[65] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1998.

[66] Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 97–107, 1996.

[67] Jacob R. Lorch and Alan Jay Smith. Reducing Processor Power Consumption by Improving Processor Time Management in a Single-User Operating System. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 143–154, 1996.

[68] Giovanna Di Marzo, Murhimanya Muhugusa, Christian Tschudin, and Jürgen Harms. The Messenger Paradigm and its Implications on Distributed Systems. In *Workshop on Intelligent Computer Communications*, 1995. Available at `http://cuisun9.unige.ch/Telecom-group/msgr/msgr.html`.

[69] Robin Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes Part I. Technical report, University of Edinburgh, 1989. ECS-LFCS-89-85.

[70] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1991.

[71] Luc Moreau. An Operational Semantics for a Parallel Functional Language with Continuations. In *PARLE'92 Parallel Architectures and Languages Europe*, volume 605 of *Lecture Notes in Computer Science*, pages 415–430, 1992.

[72] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract Models of Memory Management. In *Conference Record of FPCA'95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, 1995.

[73] George C. Necula. Proof-Carrying Code. In *Conference Record of POPL'97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[74] Rocco De Nicola, GianLuigi Ferrari, and Rosario Pugliese. Locality based Linda: programming with explicit localities. In *TAPSOFT'97*, Lecture Notes in Computer Science, 1997.

[75] Shin-ya Nishizaki. Type Inference Algorithm of Simply-Typed Lambda-Calculus with First-Class Environments. In *10th Conference Proceedings Japan Society for Software Science and Technology*, 1993.

[76] Voyager core package technical overview, 1997. ObjectSpace Inc. Available at `http://www.objectspace.com/Voyager/`.

[77] Luiz A. G. Oliveira, Paulo C. Oliveira, and Eleri Cardozo. An Agent-Based Approach for Quality of Service Negotiation and Management in Distributed Multimedia Systems. In *Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 1–12, 1997.

[78] Charles E. Perkins and David B. Johnson. Mobility Support in IPv6. In *MOBICOM'96 Proceedings of The Second Annual International Conference on Mobile Computing and Networking*, pages 27–37, 1996.

[79] Christian Queinnec. A Concurrent and Distributed Extension of Scheme. In *PARLE'92 Parallel Architectures and Languages Europe*, volume 605 of *Lecture Notes in Computer Science*, pages 431–446, 1992.

[80] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. Technical report, University of Maryland, 1996.

[81] James Riely and Matthew Hennessy. Distributed Processes and Location Failures. Technical report, University of Sussex, 1997. Report 2/97.

[82] James Riely and Matthew Hennessy. A Typed Language for Distributed Mobile Processes. In *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–390, 1998.

[83] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Coordination Languages and Models*, Lecture Notes in Computer Science, pages 211–226, 1999.

[84] Tatsurou Sekiguchi and Akinori Yonezawa. A Calculus with Code Mobility. In H. Bowman and J. Derrick, editors, *Proceedings of Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, pages 21–36. Chapman & Hall, 1997.

[85] Peter Sewell. Global/Local Subtyping for a Distributed π-calculus. Technical Report 435, Computer Laboratory, University of Cambridge, 1997.

[86] Peter Sewell, Pawl T. Wojciechowski, and Benjamin C. Pierce. Location Independence for Mobile Agents. Workshop on Internet Programming Language, 1998.

[87] Kazuyuki Shudo. Thread Migration on Java environment. Master's thesis, University of Waseda, 1997.

[88] J. M. Spivey. *The Z Notation, A Reference Manual.* International Series in Computer Science. Prentice Hall, 1989.

[89] James W. Stamos and David K. Gifford. Remote Evaluation. In *ACM Transactions on Programming Languages and Systems*, volume 12(4), pages 537–565, 1990.

[90] Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. In *Conference Record of POPL'98: 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, 1998.

[91] C. Strachey and C. P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. Technical Report PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

[92] Markus Straßer, Joachim Baumann, and Fritz Hohl. Mole – A Java based Mobile Agent System. *ECOOP'96 Workshop on Mobile Object Systems*, 1996.

[93] JavaGo, 1998. `http://web.yl.is.s.u-tokyo.ac.jp/amo/`.

[94] Tommy Thorn. Programming Languages for Mobile Code. In *ACM Computing Surveys*, volume 29(3), pages 213–239, 1997.

[95] Cristian Ungureanu and Benjamin Goldberg. Formal Models of Distributed Memory Management. Technical report, Computer Science Department of New York University, 1996. TR1996-728.

[96] Cristian Ungureanu and Benjamin Goldberg. Formal Models of Distributed Memory Management. In *Proceedings of the ACM International Conference on Functional Programming*, pages 280–291, 1997.

[97] Jan Vitek, Manuel Serrano, and Dimitri Thanos. Security and Communication in Mobile Object Systems. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 177–199, 1996.

[98] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64, 1997.

[99] Takuo Watanabe. Mobile Code Description using Partial Continuations: Definition and Operational Semantics. In *Proceedings of WOOC'97*, 1997.

[100] James E. White. Telescript Technology: An Introduction to the Language, 1995. General Magic white paper.

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

[102] XMOVE: A Pseudoserver for X Window Movement, 1994. Incorporated in X Window System.