Mobile Ambients

Luca Cardelli*
Digital Equipment Corporation
Systems Research Center

Andrew D. Gordon*
University of Cambridge
Computer Laboratory

Abstract
We introduce a calculus describing the movement of processes and devices, including movement through administrative domains.

1 Introduction

There are two distinct areas of work in mobility: *mobile computing*, concerning computation that is carried out in mobile devices (laptops, personal digital assistants, etc.), and *mobile computation*, concerning mobile code that moves between devices (applets, agents, etc.). We aim to describe all these aspects of mobility within a single framework that encompasses mobile *agents*, the *ambients* where agents interact and the mobility of the ambients themselves.

The inspiration for this work comes from the potential for mobile computation over the World-Wide Web. The geographic distribution of the Web naturally calls for mobility of computation, as a way of flexibly managing latency and bandwidth. Because of recent advances in networking and language technology, the basic tenets of mobile computation are now technologically realizable. The high-level software architecture potential, however, is still largely unexplored.

The main difficulty with mobile computation on the Web is not in mobility per se, but in the handling of *administrative domains*. In the early days of the Internet one could rely on a flat name space given by IP addresses; knowing the IP address of a computer would very likely allow one to talk to that computer in some way. This is no longer the case: firewalls partition the Internet into administrative domains that are isolated from each other except for rigidly controlled pathways. System administrators enforce policies about what can move through firewalls and how.

Mobility requires more than the traditional notion of authorization to run or to access information in certain domains: it involves the authorization to enter or exit certain domains. In particular, as far as mobile computation is concerned, it is not realistic to imagine that an agent can migrate from any point A to any point B on the Internet. Rather, an agent must first exit its administrative domain (obtaining permission to do so), enter someone else's administrative domain (again, obtaining permission to do so) and then enter a protected area of some machine where it is allowed to run (after obtaining permission to do so). Access to information is controlled at many levels, thus multiple levels of authorization may be involved. Among these levels we have: local computer, local area network, regional area network, wide-area intranet and internet. Mobile programs must be equipped to navigate this hierarchy of administrative domains, at every

* Current affiliation: Microsoft Research.

step obtaining authorization to move further. Similarly, laptops must be equipped to access resources depending on their location in the administrative hierarchy. Therefore, at the most fundamental level we need to capture notions of locations, of mobility and of authorization to move.

With these motivations, we adopt a paradigm of mobility where computational ambients are hierarchically structured, where agents are confined to ambients and where ambients move under the control of agents. A novelty of this approach is in allowing the movement of self-contained nested environments that include data and live computation, as opposed to the more common techniques that move single agents or individual objects. Our goal is to make mobile computation scale-up to widely distributed, intermittently connected and well administered computational environments.

This paper is organized as follows. In the rest of Section 1 we introduce our basic concepts and we compare them to previous and current work. In Section 2 we describe a calculus based exclusively on mobility primitives, and we use it to represent basic notions such as numerals and Turing machines, and to code a firewall-crossing protocol. In Section 3 we extend our calculus with local communication, and we show how we can represent more general communication mechanisms as well as the π-calculus.

1.1 Ambients

Ambients have the following main characteristics.

An ambient is a *bounded* placed where computation happens. The interesting property here is the existence of a boundary around an ambient. If we want to move computations easily we must be able to determine what should move; a boundary determines what is inside and what is outside an ambient. Examples of ambients, in this sense, are: a web page (bounded by a file), a virtual address space (bounded by an addressing range), a Unix file system (bounded within a physical volume), a single data object (bounded by "self") and a laptop (bounded by its case and data ports). Non-examples are: threads (where the boundary of what is "reachable" is difficult to determine) and logically related collections of objects. We can already see that a boundary implies some flexible addressing scheme that can denote entities across the boundary; examples are symbolic links, Uniform Resource Locators and Remote Procedure Call proxies. Flexible addressing is what enables, or at least facilitates, mobility. It is also, of course, a cause of problems when the addressing links are "broken".

An ambient can be nested within other ambients. As we discussed, administrative domains are (often) organized hierarchically. If we want to move a running application from work to home, the application must be removed from an enclosing (work) ambient and inserted into another enclosing (home) ambient. A laptop may need a removal pass to leave a workplace, and a government pass to leave or enter a country.

An ambient can be moved as a whole. If we move a laptop to a different network, all the address spaces and file systems within it move accordingly. If we move an agent from one computer to another, its local data moves accordingly.

Each ambient has a name that is used to control access to the ambient. A name is something that can be created and passed around, and from which access capabilities can be extracted. In a realistic situation the true name of an ambient would be guarded very closely, and only specific capabilities would be handed out.

1.2 Technical Context: Systems

Many software systems have explored and are exploring notions of mobility.

Obliq [5] attacks the problems of distribution and mobility for intranet computing. Obliq works well for its intended application, but is not really suitable for computation and mobility over the Web (like other distributed paradigms based on the remote procedure call model) because of the fragility of network proxies over the Web.

Our ambient model is partially inspired by Telescript [16], but is almost dual to it. In Telescript, agents move whereas places stay put. Ambients, instead, move whereas agents are confined to ambients. A Telescript agent, however, is itself a little ambient, since it contains a "suitcase" of data. Some nesting of places is allowed in Telescript.

Java [11] provides a working framework for mobile computation, as well as a widely available infrastructure on which to base more ambitious mobility efforts.

Linda [6] is a "coordination language" where multiple processes interact in a common space (called a tuple space) by exchanging tokens asynchronously. Distributed versions of Linda exist that use multiple tuple spaces and allow remote operations. A dialect of Linda [7] allows nested tuple spaces, but not mobility of the tuple spaces.

1.3 Technical Context: Formalisms

Many existing calculi have provided inspiration for our work.

The π-calculus [15] is a process calculus where channels can "move" along other channels. The movement of processes is represented as the movement of channels that refer to processes. Therefore, there is no clear indication that processes themselves move. For example, if a channel crosses a firewall (that is, if it is communicated to a process meant to represent a firewall), there is no clear sense in which the process has also crossed the firewall. In fact, the channel may cross several independent firewalls, but a process could not be in all those places at once. Nonetheless, many fundamental π-calculus concepts and techniques underlie our work.

The spi calculus [1] extends the π-calculus with cryptographic primitives. The need for such extensions does not seem to arise immediately within our ambient calculus. Some of the motivations for the spi calculus extension are already covered by the notion of encapsulation within an ambient. However, we do not know yet how extensively we can use our ambient primitives for cryptographic purposes.

The Chemical Abstract Machine [3] is a semantic framework, rather than a specific formalism. Its basic notions of reaction in a solution and of membranes that isolate subsolutions, closely resemble ambient notions. However, membranes are not meant to provide strong protection, and there is no concern for mobility of subsolutions. Still, we adopt a "chemical style" in presenting our calculus.

The join-calculus [9] is a reformulation of the π-calculus with a more explicit notion of places of interaction; this greatly helps in building distributed implementations of channel mechanisms. The distributed join-calculus [10] adds a notion of named locations, with essentially the same aims as ours, and a notion of distributed failure. Locations in the distributed join-calculus form a tree, and subtrees can migrate from one part of the tree to another. A main difference with our ambients is that movement may happen directly from any active location to any other known location.

LLinda [8] is a formalization of Linda using process calculi techniques. As in dis-

tributed versions of Linda, LLinda has multiple distributed tuple spaces. Multiple tuple spaces are very similar in spirit to multiple ambients, but Linda's tuple spaces do not nest, and there are no restrictions about accessing a tuple space from another one.

Finally, a growing body of literature is concentrating on the idea of adding discrete locations to a process calculus and considering failure of those locations [2, 10]. Our notion of locality is built into our basic calculus. It is induced by a non-trivial and dynamic topology of locations, in the sense that a location that is "far" from the current one can only be reached through multiple individual moves. Failure of a location can be represented as becoming forever unreachable.

2 Mobility

We begin by describing a minimal calculus of ambients that includes only mobility primitives. Still, we shall see that this calculus is quite expressive. In Section 3 we then add communication primitives.

2.1 Mobility Primitives

The syntax of the calculus is defined in the following table. The main syntactic categories are processes (including ambients and agents that execute actions) and capabilities.

Mobility Primitives

P,Q ::=	processes	n	names
$(\nu n)P$	restriction		
$\mathbf{0}$	inactivity	M ::=	capabilities
$P \mid Q$	composition	*in* n	can enter n
$!P$	replication	*out* n	can exit n
$n[P]$	ambient	*open* n	can open n
$M.P$	action		

Syntactic conventions

$(\nu n)P \mid Q$	$= ((\nu n)P) \mid Q$	$(\nu n_1 ... n_m)P$	$\triangleq$	$(\nu n_1)...(\nu n_m)P$	
$!P \mid Q$	$= (!P) \mid Q$	$n[]$	$\triangleq$	$n[\mathbf{0}]$	
$M.P \mid Q$	$= (M.P) \mid Q$	M	$\triangleq$	$M.\mathbf{0}$	(where appropriate)

The first four process primitives (restriction, inactivity, composition and replication) have the same meaning as in the π-calculus (see Section 2.3), namely: restriction is used to introduce new names and limit their scope; $\mathbf{0}$ has no behavior; $P \mid Q$ is the parallel composition of P and Q; and $!P$ is an unbounded number of parallel replicas of P. The main difference with respect to the π-calculus is that names are used to name ambients instead of channels. To these standard primitives we add ambients, $n[P]$, and the exercise of capabilities, $M.P$. Next we discuss these new primitives in detail.

2.2 Explanations

We begin by introducing the semantics of ambients informally. A reduction relation $P \longrightarrow Q$ describes the evolution of a process P into a new process Q.

Ambients

An ambient is written $n[P]$, where n is the name of the ambient, and P is the process running inside the ambient. In $n[P]$, it is understood that P is actively running, and that P can be the parallel composition of several processes. We emphasize that P is running even when the surrounding ambient is moving. Running while moving may or may not be realistic, depending on the nature of the ambient and of the communication medium through which the ambient moves, but it is consistent to think in those terms. We express the fact that P is running by a rule that says that any reduction of P becomes a reduction of $n[P]$:

$$P \longrightarrow Q \;\Rightarrow\; n[P] \longrightarrow n[Q]$$

In general, an ambient exhibits a tree structure induced by the nesting of ambient brackets. Each node of this tree structure may contain a collection of (non-ambient) processes running in parallel, in addition to subambients. We say that these processes are running in the ambient, in contrast to the ones running in subambients.

Nothing prevents the existence of two or more ambients with the same name, either nested or at the same level. Once a name is created, it can be used to name multiple ambients. Moreover, $!n[P]$ generates multiple ambients with the same name. This way, for example, one can easily model the replication of services.

Actions and Capabilities

Operations that change the hierarchical structure of ambients are sensitive. In particular such operations can be interpreted as the crossing of firewalls or the decoding of ciphertexts. Hence these operations are restricted by *capabilities*. Thanks to capabilities, an ambient can allow other ambients to perform certain operations without having to reveal its true name. With the communication primitives of Section 3, capabilities can be transmitted as values.

The process $M.\ P$ executes an action regulated by the capability M, and then continues as the process P. The process P does not start running until the action is executed. The reduction rules for $M.\ P$ depend on the capability M, and are described below case by case.

We consider three kinds of capabilities: one for entering an ambient, one for exiting an ambient and one for opening up an ambient. Capabilities are obtained from names; given a name n, the capability *in* n allows entry into n, the capability *out* n allows exit out of n and the capability *open* n allows the opening of n. Implicitly, the possession of one or all of these capabilities for n is insufficient to reconstruct the original name n.

An entry capability, *in* m, can be used in the action *in* $m.\ P$, which instructs the ambient surrounding *in* $m.\ P$ to enter a sibling ambient named m. If no sibling m can be found, the operation blocks until a time when such a sibling exists. If more than one m sibling exists, any one of them can be chosen. The reduction rule is:

$$n[in\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$$

If successful, this reduction transforms a sibling n of an ambient m into a child of m. After the execution, the process *in* $m.\ P$ continues with P, and both P and Q find themselves at a lower level in the tree of ambients.

An exit capability, *out m*, can be used in the action *out m*. *P*, which instructs the ambient surrounding *out m*. *P* to exit its parent ambient named *m*. If the parent is not named *m*, the operation blocks until a time when such a parent exists. The reduction rule is:

$$m[n[\mathit{out}\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$$

If successful, this reduction transforms a child *n* of an ambient *m* into a sibling of *m*. After the execution, the process *in m*. *P* continues with *P*, and both *P* and *Q* find themselves at a higher level in the tree of ambients.

An opening capability, *open n*, can be used in the action *open n*. *P*. This action provides a way of dissolving the boundary of an ambient named *n* located at the same level as *open*, according to the rule:

$$\mathit{open}\ n.\ P \mid n[Q] \longrightarrow P \mid Q$$

If no ambient *n* can be found, the operation blocks until a time when such an ambient exists. If more than one ambient *n* exists, any one of them can be chosen.

An *open* operation may be upsetting to both *P* and *Q* above. From the point of view of *P*, there is no telling in general what *Q* might do when unleashed. From the point of view of *Q*, its environment is being ripped open. Still, this operation is relatively well-behaved because: (1) the dissolution is initiated by the agent *open n*. *P*, so that the appearance of *Q* at the same level as *P* is not totally unexpected; (2) *open n* is a capability that is given out by *n*, so *n*[*Q*] cannot be dissolved if it does not wish to be.

Movement from the Inside or the Outside: Subjective vs. Objective

There are two natural kinds of movement primitives for ambients. The distinction is between "I make you move" from the outside (*objective move*) or "I move" from the inside (*subjective move*). Subjective moves have been described above. Objective moves (indicated by an *mv* prefix), obey the rules:

$$\mathit{mv}\ \mathit{in}\ m.\ P \mid m[R] \longrightarrow m[P \mid R] \qquad\qquad m[\mathit{mv}\ \mathit{out}\ m.\ P \mid R] \longrightarrow P \mid m[R]$$

These two kinds of move operations are not trivially interdefinable. The objective moves have simpler rules. However, they operate only on ambients that are not active; they provide no way of moving an existing running ambient. The subjective moves, in contrast, cause active ambients to move and, together with *open*, can approximate the effect of objective moves (as we discuss later).

In evaluating these alternative operations, one should consider who has the authority to move whom. In general, the authority to move rests in the top-level agents of an ambient, which naturally act as *control agents*. Control agents cannot be injected purely by subjective moves, since these moves handle whole ambients. With objective moves, instead, a control agent can be injected into an ambient simply by possessing an entry capability for it. As a consequence, objective moves and entry capabilities together provide the unexpected power of entrapping an ambient into a location it can never exit:

$$\mathit{entrap}\ m \triangleq (\nu\ k)\ (k[] \mid \mathit{mv}\ \mathit{in}\ m.\ \mathit{in}\ k.\ \mathbf{0})$$
$$\mathit{entrap}\ m \mid m[P] \longrightarrow^{*} (\nu k)\ k[m[P]]$$

The *open* capability confers the right to dissolve an ambient from the outside and reveal its contents. It is interesting to consider an operation that dissolves an ambient form the inside, called *acid*:

$$m[acid.\ P \mid Q] \longrightarrow P \mid Q$$

Acid gives a simple encoding of objective moves:

$$mv\ in\ n.P \triangleq (\nu q)\ q[in\ n.\ acid.\ P]$$
$$mv\ out\ n.P \triangleq (\nu q)\ q[out\ n.\ acid.\ P]]$$

Therefore, *acid* is as dangerous as objective moves, providing the power to entrap ambients. We shall see that *open* can be used to define a capability-restricted version of *acid* that does not lead to entrapment.

2.3 Operational Semantics

We now give an operational semantics of the calculus of section 2.1, based on a structural congruence between processes, $\equiv$, and a reduction relation $\longrightarrow$. This is a semantics in the style of Milner's reaction relation [14] for the π-calculus, which was itself inspired by the Chemical Abstract Machine of Berry and Boudol [3].

Structural Congruence

$P \equiv P$	$P \mid Q \equiv Q \mid P$
$P \equiv Q \Rightarrow Q \equiv P$	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	$!P \equiv P \mid !P$
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ if $n \notin fn(P)$
$P \equiv Q \Rightarrow !P \equiv !Q$	$(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $n \neq m$
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	$P \mid \mathbf{0} \equiv P$
$P \equiv Q \Rightarrow M.P \equiv M.Q$	$(\nu n)\mathbf{0} \equiv \mathbf{0}$
	$!\mathbf{0} \equiv \mathbf{0}$

Processes of the calculus are grouped into equivalence classes by the relation $\equiv$, which denotes structural congruence (that is, equivalence up to trivial syntactic restructuring). In addition, we identify processes up to renaming of bound names: $(\nu n)P = (\nu m)P\{n \leftarrow m\}$ if $m \notin fn(P)$. By this we mean that these processes are understood to be identical (for example, by choosing an appropriate representation), as opposed to structurally equivalent.

Note that the following terms are in general distinct:

$!(\nu n)P \not\equiv (\nu n)!P$	replication creates new names
$n[P] \mid n[Q] \not\equiv n[P \mid Q]$	multiple n ambients have separate identity

The behavior of processes is given by the following reduction relations. The first three rules are the one-step reductions for *in*, *out* and *open*. The next three rules propagate reductions across scopes, ambient nesting and parallel composition. The final rule allows the use of equivalence during reduction. Finally, $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

Reduction

$n[in\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$ $\quad P \longrightarrow Q \ \Rightarrow\ (\nu n)P \longrightarrow (\nu n)Q$

$m[n[out\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$ $\quad P \longrightarrow Q \ \Rightarrow\ n[P] \longrightarrow n[Q]$

$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$ $\quad P \longrightarrow Q \ \Rightarrow\ P \mid R \longrightarrow Q \mid R$

$P' \equiv P,\ P \longrightarrow Q,\ Q \equiv Q' \ \Rightarrow\ P' \longrightarrow Q'$

2.4 Example: Locks

We can use *open* to encode locks that are released and acquired:

$$acquire\ n.\ P \triangleq open\ n.\ P \qquad release\ n.\ P \triangleq n[] \mid P$$

This way, two agents can "shake hands" before proceeding with their execution:

$$acquire\ n.\ release\ m.\ P \mid release\ n.\ acquire\ m.\ Q$$

2.5 Example: Firewall Access

In this example, an agent crosses a firewall by means of previously arranged passwords k, k', and k''. The agent exhibits the password k' by using a wrapper ambient that has k' as its name. The firewall, which has a secret name w, sends out a pilot ambient, $k[out\ w.\ in\ k'.\ in\ w]$, to guide the agent inside. The pilot ambient enters an agent by performing $in\ k'$ (therefore verifying that the agent knows the password), and is given control by being opened. Then, $in\ w$ transports the agent inside the firewall, where the password wrapper is discarded. The third name, k'', is needed to confine the contents Q of the agent and to prevent Q from interfering with the protocol.

The final effect is that the agent physically crosses into the firewall; this can be seen below by the fact that Q is finally placed inside w. (For simplicity, this example is written to allow a single agent to enter.) Assume $(fn(P) \cup fn(Q)) \cap \{k, k', k''\} = \emptyset$ and $w \notin fn(Q)$:

$$\begin{aligned} Firewall &\triangleq (\nu w)\ w[k[out\ w.\ in\ k'.\ in\ w] \mid open\ k'.\ open\ k''.\ P] \\ Agent &\triangleq k'[open\ k.\ k''[Q]] \end{aligned}$$

There is no guarantee here that any particular agent will make it inside the firewall. Rather, the intended guarantee is that if any agent crosses the firewall, it must be one that knows the passwords.

To express the security property of the firewall we introduce a notion of contextual equivalence, $\simeq$. Let a context $\boldsymbol{C}[]$ be a process containing zero or more holes, and for any process P, let $\boldsymbol{C}[P]$ be the process obtained by filling each hole in $\boldsymbol{C}$ with a copy of P (names free in P may become bound). Then define:

$$\begin{aligned} P{\downarrow}n &\triangleq P \equiv (\nu\, m_1 ... m_i)\ (n[P'] \mid P'') \qquad \text{where } n \notin \{m_1 ... m_i\} \\ P{\Downarrow}n &\triangleq P \longrightarrow^* Q \text{ and } Q{\downarrow}n \\ P \simeq Q &\triangleq \text{for all } n \text{ and } \boldsymbol{C}[],\ \boldsymbol{C}[P]{\Downarrow}n \Leftrightarrow \boldsymbol{C}[Q]{\Downarrow}n \end{aligned}$$

If $(fn(P) \cup fn(Q)) \cap \{k, k', k''\} = \emptyset$ and $w \notin fn(Q)$, then we can show that the interaction of the agent with the firewall produces the desired result up to contextual equivalence.

$$(\nu\ k\ k'\ k'')\ (Agent \mid Firewall) \ \simeq\ (\nu w)\ w[Q \mid P]$$

Since contextual equivalence takes into account all possible contexts, the equation above states that the firewall crossing protocol works correctly in the presence of any possible attacker (that does not know the passwords) that may try to disrupt it.

2.6 Example: Objective Moves and Dissolution

Objective moves are not directly encodable. However, specific ambients can explicitly allow objective moves by using *open*:

$$\begin{array}{lll}
allow\ n & \triangleq & !open\ n \\
mv\ in\ n.P & \triangleq & (\nu k)\ k[in\ n.\ in[out\ k.\ open\ k.\ P]] \\
mv\ out\ n.P & \triangleq & (\nu k)\ k[out\ n.\ out[out\ k.\ open\ k.\ P] \\
n^{\downarrow}[P] & \triangleq & n[P \mid allow\ in] \qquad (n^{\downarrow} \text{ allows } mv\ in) \\
n^{\uparrow}[P] & \triangleq & n[P] \mid allow\ out \qquad (n^{\uparrow} \text{ allows } mv\ out) \\
n^{\downarrow\uparrow}[P] & \triangleq & n[P \mid allow\ in] \mid allow\ out \qquad (n^{\downarrow\uparrow} \text{ allows both } mv\ in \text{ and } mv\ out)
\end{array}$$

These definitions are to be used, for example, as follows:

$$\begin{array}{l}
mv\ in\ n.P \mid n^{\downarrow\uparrow}[Q\ ] \ \longrightarrow^*\ n^{\downarrow\uparrow}[P \mid Q] \\
n^{\downarrow\uparrow}[mv\ out\ n.P \mid Q] \ \longrightarrow^*\ P \mid n^{\downarrow\uparrow}[Q]
\end{array}$$

Similarly, the *acid* primitive discussed previously is not encodable via *open*. However, we can code a form of planned dissolution:

$$acid\ n.\ P \ \triangleq\ acid[out\ n.\ open\ n.\ P]$$

to be used with a helper process *open acid* as follows:

$$n[acid\ n.\ P \mid Q] \mid open\ acid \ \longrightarrow^*\ P \mid Q$$

This form of *acid* is sufficient for uses in many encodings where it is necessary to dissolve ambients. Encodings are carefully planned, so it is easy to add the necessary *open* instructions. The main difference with the liberal form of *acid* is that *acid n* must name the ambient it is dissolving. More precisely, the encoding of *acid n* requires both an exit and an open capability for *n*.

2.7 Example: External Choice

A major feature of CCS [13] is the presence of a non-deterministic choice operator (+). We do not take + as a primitive, in the spirit of the asynchronous π-calculus, but we can approximate some aspects of it by the following definitions. The intent is that $n \Rightarrow P + m \Rightarrow Q$ reduces to P in the presence of an n ambient, and reduces to Q in the presence of an m ambient.

$$\begin{array}{lll}
n \Rightarrow P + m \Rightarrow Q & \triangleq & (\nu\ p\ q\ r)\ (\\
 & & \quad p[in\ n.\ out\ n.\ q[out\ p.\ open\ r.\ P]] \mid \\
 & & \quad p[in\ m.\ out\ m.\ q[out\ p.\ open\ r.\ Q]] \mid \\
 & & \quad open\ q \mid r[])
\end{array}$$

For example, assuming $\{p, q, r\} \cap fn(R) = \emptyset$, we have:

$$(n \Rightarrow P + m \Rightarrow Q) \mid n[R] \quad \longrightarrow^* \simeq \quad P \mid n[R]$$

where the relation $\longrightarrow^* \simeq$ is the relational composition of $\longrightarrow^*$ and $\simeq$.

2.8 Example: Numerals

We represent the number i by a stack of nested ambients of depth i. For any natural number i, let $\underline{i}$ be the numeral for i:

$$\underline{0} \triangleq zero[] \qquad\qquad \underline{i+1} \triangleq succ[open\ op \mid \underline{i}]$$

The *open op* process is needed to allow ambients named *op* to enter the stack of ambients to operate on it. To show that arithmetic may be programmed on these numerals, we begin with an *ifzero* operation to tell whether a numeral represents 0 or not.

$$ifzero\ P\ Q \triangleq zero \Rightarrow P + succ \Rightarrow Q$$
$$\underline{0} \mid ifzero\ P\ Q \longrightarrow^* \simeq \underline{0} \mid P$$
$$\underline{i+1} \mid ifzero\ P\ Q \longrightarrow^* \simeq \underline{i+1} \mid Q$$

Next, we can encode increment and decrement operations.

$$inc.P \triangleq ifzero\ (inczero.P)\ (incsucc.P)$$
$$inczero.P \triangleq open\ zero.\ (\underline{1} \mid P)$$
$$incsucc.P \triangleq (\nu\ p\ q)\ (p[succ[open\ op]] \mid open\ q.\ open\ p.\ P \mid op[in\ succ.\ in\ p.\ in\ succ.\ (q[out\ succ.\ out\ succ.\ out\ p] \mid open\ op)])$$
$$dec.P \triangleq (\nu\ p)\ (op[in\ succ.\ p[out\ succ]] \mid open\ p.\ open\ succ.\ P)$$

These definitions satisfy:

$$\underline{i} \mid inc.P \longrightarrow^* \simeq \underline{i+1} \mid P \qquad\qquad \underline{i+1} \mid dec.P \longrightarrow^* \simeq \underline{i} \mid P$$

Given that iterative computations can be programmed with replication, any arithmetic operation can be programmed with *inc*, *dec* and *iszero.*

2.9 Example: Turing Machines

We emulate Turing machines in a "mechanical" style. A tape consists of a nested sequence of squares, each initially containing the flag *ff*[]. The first square has a distinguished name to indicate the end of the tape to the left:

$$end^{\downarrow\uparrow}[ff[] \mid sq^{\downarrow\uparrow}[ff[] \mid sq^{\downarrow\uparrow}[ff[] \mid sq^{\downarrow\uparrow}[ff[] \mid ...\ ]]]]$$

The head of the machine is an ambient that inhabits a square. The head moves right by entering the next nested square and moves left by exiting the current square. The head contains the program of the machine and it can read and write the flag in the current square. The trickiest part of the definition concerns extending the tape. Two tape-stretchers are placed at the beginning and end of the tape and continuously add squares.

$$if\ tt\ P, if\ ff\ Q \triangleq tt \Rightarrow open\ tt.\ P + ff \Rightarrow open\ ff.\ Q$$

$$head \triangleq$$
$$head[!open\ S_1. \qquad \text{state \#1 (example)}$$

$mv\ out\ head.$ — jump out to read flag
$\quad if\ tt\ (ff[] \mid mv\ in\ head.\ in\ sq.\ S_2[]),$ — head right, state #2
$\quad if\ ff\ (tt[] \mid mv\ in\ head.\ out\ sq.\ S_3[])\ \mid$ — head left, state #3
$\ldots \mid$ — more state transitions
$S_1[]$ — initial state

$stretchRht \triangleq$ — stretch tape right

$(\nu r)\ r[!open\ it.\ mv\ out\ r.\ (sq^{\downarrow\uparrow}[ff[]] \mid mv\ in\ r.\ in\ sq.\ it[]) \mid it[]]$

$stretchLft \triangleq$ — stretch tape left

$!open\ it.\ mv\ in\ end.$
$\quad (mv\ out\ end.\ end^{\downarrow\uparrow}[sq^{\downarrow\uparrow}[] \mid ff[]] \mid$
$\quad\ in\ end.\ in\ sq.\ mv\ out\ end.\ open\ end.\ mv\ out\ sq.\ mv\ out\ end.\ it[])$
$\mid it[]$

$machine \triangleq stretchLft \mid end^{\downarrow\uparrow}[ff[] \mid head \mid stretchRht]$

3 Communication

Although the pure mobility calculus is powerful enough to be Turing-complete, it has no communication or variable-binding operators. Such operators seem necessary, for example, to comfortably encode other formalisms such as the π-calculus.

Therefore, we now have to choose a communication mechanism to be used to exchange messages between ambients. The choice of a particular mechanism is somewhat orthogonal to the mobility primitives. However, we should try not to defeat with communication the restrictions imposed by capabilities. This suggests that a primitive form of communication should be purely local, and that the transmission of non-local messages should be restricted by capabilities.

3.1 Communication Primitives

To focus our attention, we pose as a goal the ability to encode the asynchronous π-calculus. For this it is sufficient to introduce a simple asynchronous communication mechanism that works locally within a single ambient.

Mobility and Communication Primitives

P,Q ::=	processes	M ::=	capabilities
$(\nu n)P$	restriction	x	variable
$\mathbf{0}$	inactivity	n	name
$P \mid Q$	composition	$in\ M$	can enter into M
$!P$	replication	$out\ M$	can exit out of M
$M[P]$	ambient	$open\ M$	can open M
$M.P$	capability action	ε	null
$(x).P$	input action	$M.M'$	path
$\langle M \rangle$	async output action		

We again start by displaying the syntax of a whole calculus. The mobility primi-

tives are essentially those of section 2, but the addition of communication variables changes some of the details. More interestingly, we add input ($(x).P$) and output ($\langle M\rangle$) primitives and we enrich the capabilities to include paths. We identify capabilities up to the following equations: $L.(M.N) = (L.M).N$ and $M.\varepsilon = M = \varepsilon.M$. As a new syntactic convention, we have that $(x).P \mid Q = ((x).P) \mid Q$.

3.2 Explanations

Communicable Values

The entities that can be communicated are either names or capabilities. In realistic situations, communication of names should be rather rare, since knowing the name of an ambient gives a lot of control over it. Instead, it should be common to communicate restricted capabilities to allow controlled interactions between ambients.

It now becomes useful to combine multiple capabilities into *paths*, especially when one or more of those capabilities are represented by input variables. To this end we introduce a path-formation operation on capabilities ($M.\ M'$). For example, ($in\ n.\ in\ m$). P is interpreted as $in\ n.\ in\ m.\ P$.

We distinguish between ν-bound names and input-bound variables. Variables can be instantiated with names or capabilities. In practice, we do not need to distinguish these two sorts lexically, but we often use n, m, p, q for names and w, x, y, z for variables.

Ambient I/O

The simplest communication mechanism that we can imagine is local anonymous communication within an ambient (ambient I/O, for short):

$(x).P$ input action $\langle M\rangle$ async output action

An output action releases a capability (possibly a name) into the local ether of the surrounding ambient. An input action captures a capability from the local ether and binds it to a variable within a scope. We have the reduction:

$$(x).P \mid \langle M\rangle \longrightarrow P\{x \leftarrow M\}$$

This local communication mechanism fits well with the ambient intuitions. In particular, long-range communication, like long-range movement, should not happen automatically because messages may have to cross firewalls. Still, this simple mechanism is sufficient, as we shall see, to emulate communication over named channels, and more generally to provide an encoding of the asynchronous π-calculus.

Remark

To allow both names and capabilities to be output and input, there is a single syntactic sort that includes both. Then, a meaningless term of the form $n.\ P$ can then arise, for instance, from the process $((x).\ x.\ P) \mid \langle n\rangle$. This anomaly is caused by the desire to denote movement capabilities by variables, as in $(x).\ x.\ P$, and from the desire to denote names by variables, as in $(x).\ x[P]$. We permit $n.\ P$ to be formed, syntactically, in order to make substitution always well defined. A simple type system distinguishing names from movement capabilities would avoid this anomaly.

3.3 Operational Semantics

The structural congruence relation is defined as in section 2.3, with the understanding that P and M range now over larger classes, and with the addition of the following equivalences:

Structural Congruence

$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	$\varepsilon.P \equiv P$
$P \equiv Q \Rightarrow (x).P \equiv (x).Q$	$(M.M').P \equiv M.M'.P$

We now identify processes up to renaming of bound variables: $(x).P = (y).P\{x \leftarrow y\}$ if $y \notin fv(P)$. Finally, we have a new reduction rule:

Reduction

$$(x).P \mid \langle M \rangle \longrightarrow P\{x \leftarrow M\}$$

3.4 Example: Cells

A cell *cell c w* stores a value *w* at a location *c*, where a value is a capability. The cell is set to output its current contents destructively, and is set to be "refreshed" with either the old contents (by *get*) or a new contents (by *set*). Note that *set* is essentially an output operation, but it is a synchronous one: its sequel *P* runs only after the cell has been set. Parallel *get* and *set* operations do not interfere.

$$\begin{aligned} &\textit{cell c w} \triangleq c^{\downarrow\uparrow}[\langle w \rangle] \\ &\textit{get c}\ (x).\ P \triangleq \textit{mv in c}.\ (x).\ (\langle x \rangle \mid \textit{mv out c}.\ P) \\ &\textit{set c}\ \langle w \rangle.\ P \triangleq \textit{mv in c}.\ (x).\ (\langle w \rangle \mid \textit{mv out c}.\ P) \end{aligned}$$

It is possible to code an atomic *get-and-set* primitive:

$$\textit{get-and-set c}\ (x)\ \langle w \rangle.\ P \triangleq \textit{mv in c}.\ (x).\ (\langle w \rangle \mid \textit{mv out c}.\ P)$$

Named cells can be assembled into ambients that act as record data structures.

3.5 Example: Routable Packets and Active Networks

We define *packet pkt* as an empty packet of name *pkt* that can be routed repeatedly to various destinations. We also define *route pkt with P to M* as the act of placing *P* inside the packet *pkt* and sending the packet to *M*; this is to be used in parallel with *packet pkt*. Note that *M* can be a compound capability, representing a path to follow. Finally, *forward pkt to M* is an abbreviation that forwards any packet named *pkt* that passes by to *M*. Here we assume that *P* does not interfere with routing.

$$\begin{aligned} &\textit{packet pkt} \triangleq \textit{pkt}[!(x).\ x \mid !\textit{open route}] \\ &\textit{route pkt with P to M} \triangleq \textit{route}[\textit{in pkt}.\ \langle M \rangle \mid P] \\ &\textit{forward pkt to M} \triangleq \textit{route pkt with } \mathbf{0} \textit{ to M} \end{aligned}$$

Since our packets are ambients, they may contain behavior that becomes active within the intermediate routers. Therefore we can naturally model *active networks*, which are characterized by routers that execute code carried by packets.

3.6 Communication Between Ambients

Our basic communication primitives operate only within a given ambient. We now discuss one example of communication across ambients. In addition, in section 3.7 we treat the specific case of channel-based communication across ambients.

It is not realistic to assume direct long-range communication. Communication, like movement, is subject to access restrictions due to the existence of administrative domains. Therefore, it is convenient to model long-range communication as the movement of "messenger" agents that must cross administrative boundaries. Assume, for simplicity, that the location M allows I/O by $!\mathit{open}\ \mathit{io}$. By M^{-1} we indicate a given return path from M.

$@M\langle a\rangle$	$\triangleq\ \ \mathit{io}[M.\ \langle a\rangle]$	remote output at M
$@M(x)M^{-1}.\ P$	$\triangleq\ \ (\nu n)\ (\mathit{io}[M.\ (x).\ n[M^{-1}.\ P]]\ \mid\ \mathit{open}\ n)$	remote input at M

To avoid transmitting P all the way there and back, we can write input as:

$$@M(x)M^{-1}.\ P\ \ \triangleq\ \ (\nu n)\ (\mathit{io}[M.\ (x).\ n[M^{-1}.\ \langle x\rangle]]\ \mid\ \mathit{open}\ n)\ \mid\ (x).\ P$$

To emulate Remote Procedure Call we write (assuming *res* contains the result):

$$@M\ \mathit{arg}\langle a\rangle\ \mathit{res}(x)\ M^{-1}.\ P\ \ \triangleq$$
$$(\nu n)\ (\mathit{io}[M.\ (\langle a\rangle\ \mid\ \mathit{open}\ \mathit{res}.\ (x).\ n[M^{-1}.\ \langle x\rangle])]\ \mid\ \mathit{open}\ n)\ \mid\ (x).\ P$$

This is essentially an implementation of a synchronous communication (RPC) by two asynchronous communications ($\langle a\rangle$ and $\langle x\rangle$).

3.7 Encoding the π-calculus

The encoding of the asynchronous π-calculus is moderately easy, given our I/O primitives. A channel is simply represented by an ambient: the name of the channel is the name of the ambient. This is very similar in spirit to the join-calculus [9] where channels are rooted at a location. Communication on a channel is represented by local communication inside an ambient. The basic technique is a variation on objective moves. A conventional name, *io*, is used to transport input and output requests into the channel. The channel opens all such requests and lets them interact.

$\mathit{ch}\ n$	$\triangleq$	$n[!\mathit{open}\ \mathit{io}]$	a channel
$(\mathit{ch}\ n)P$	$\triangleq$	$(\nu n)\ (\mathit{ch}\ n\ \mid\ P)$	a new channel
$n(x).P$	$\triangleq$	$(\nu p)\ (\mathit{io}[\mathit{in}\ n.\ (x).\ p[\mathit{out}\ n.\ P]]\ \mid\ \mathit{open}\ p)$	channel input
$n\langle M\rangle$	$\triangleq$	$\mathit{io}[\mathit{in}\ n.\ \langle M\rangle]$	async channel output

These definitions satisfy the expected reduction $n(x).P\ \mid\ n\langle M\rangle \longrightarrow^{*} P\{x\leftarrow M\}$ in the presence of a channel $\mathit{ch}\ n$. Therefore, we can write the following encoding of the π-calculus:

Encoding of the Asynchronous π-calculus

$\llparenthesis(\nu n)P\rrparenthesis$	$\triangleq$	$(\nu n)\ (n[!\mathit{open}\ \mathit{io}]\ \mid\ \llparenthesis P\rrparenthesis)$	$\llparenthesis P\ \mid\ Q\rrparenthesis$	$\triangleq$	$\llparenthesis P\rrparenthesis\ \mid\ \llparenthesis Q\rrparenthesis$
$\llparenthesis n(x).P\rrparenthesis$	$\triangleq$	$(\nu p)\ (\mathit{io}[\mathit{in}\ n.\ (x).\ p[\mathit{out}\ n.\ \llparenthesis P\rrparenthesis]]\ \mid\ \mathit{open}\ p)$	$\llparenthesis !\mathrm{P}\rrparenthesis$	$\triangleq$	$!\llparenthesis \mathrm{P}\rrparenthesis$
$\llparenthesis n\langle m\rangle\rrparenthesis$	$\triangleq$	$\mathit{io}[\mathit{in}\ n.\ \langle m\rangle]$			

This encoding includes the choice-free synchronous π-calculus, since it can itself be encoded within the asynchronous π-calculus [4, 12].

We can fairly conveniently use these definitions to embed communication on named channels within the ambient calculus (provided the name *io* is not used for other purposes). Communication on these named channels, though, only works within a single ambient. In other words, from our point of view, a π-calculus process always inhabits a single ambient. Therefore, the notion of mobility in the π-calculus (communication of names over named channels) is different from our notion of mobility.

4 Conclusions and Future Work

We have introduced the informal notion of mobile ambients, and we have discussed how this notion captures the structure of complex networks and the behavior of mobile computation. We have then investigated an ambient calculus that formalizes this notion simply and powerfully. Our calculus is no more complex than common process calculi, but supports reasoning about mobility and, at least to some degree, security.

This paper concentrates mostly on examples and intuition. In ongoing work we are developing theories of equivalences for the ambient calculus, drawing on earlier work on the π-calculus. These equivalences will allow us to reason about mobile computation, as briefly illustrated in the firewall crossing example.

On this foundation, we can envision new programming methodologies, programming libraries and programming languages for global computation.

Acknowledgments

Thanks to Cédric Fournet, Paul McJones and Jan Vitek for comments on early drafts. Stuart Wray suggested an improved definition of external choice.

Gordon held a Royal Society University Research Fellowship for most of the time we worked on this paper.

References

[1] Abadi, M. and A.D. Gordon, **A calculus for cryptographic protocols: the spi calculus**. *Proc. Fourth ACM Conference on Computer and Communications Security*, 36-47, 1997.

[2] Amadio, R.M., **An asynchronous model of locality, failure, and process mobility**. *Proc. COORDINATION 97*, Berlin, 1997.

[3] Berry, G. and G. Boudol, **The chemical abstract machine**. *Theoretical Computer Science* **96**(1), 217-248, 1992.

[4] Boudol, G., **Asynchrony and the π-calculus.** *TR 1702, INRIA, Sophia-Antipolis*, 1992.

[5] Cardelli, L., **A language with distributed scope**. *Computing Systems*, **8**(1), 27-59. MIT Press. 1995.

[6] Carriero, N. and D. Gelernter, **Linda in context**. *CACM*, **32**(4), 444-458, 1989.

[7] Carriero, N., D. Gelernter, and L. Zuck, **Bauhaus Linda**, in LNCS 924, 66-76, Springer-Verlag, 1995.

[8] De Nicola, R., G.-L. Ferrari and R. Pugliese, **Locality based Linda: programming with explicit localities**. *Proc. TAPSOFT'97*. 1997.

[9] Fournet, C. and G. Gonthier, **The reflexive CHAM and the join-calculus**. *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, 372-385. 1996.

[10] Fournet, C., G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy, **A calculus of mobile agents**. *Proc. CONCUR'96*, 406-421. 1996.

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

[12] Honda., K. and M. Tokoro, **An object calculus for asynchronous communication.** *Proc. ECOOP'91*, LNCS 521, 133-147, Springer Verlag, 1991.

[13] Milner, R., **A calculus of communicating systems**. LNCS 92. Springer-Verlag. 1980.

[14] Milner, R., **Functions as processes**. *Mathematical Structures in Computer Science* **2**, 119-141. 1992.

[15] Milner, R., J. Parrow and D. Walker, **A calculus of mobile processes, Parts 1-2**. *Information and Computation*, **100**(1), 1-77. 1992

[16] White, J.E., **Mobile agents**. In *Software Agents*, J. Bradshaw, ed. AAAI Press / The MIT Press. 1996.