Mobile Ambients

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
<http://www.research.digital.com/SRC/personal/Luca_Cardelli/home.html>

Andrew D. Gordon

University of Cambridge, Computer Laboratory
<http://www.cl.cam.ac.uk/users/adg>

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 ob-

taining 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 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.

Today, it is very difficult to transport a working environment between two computers, for example, between a laptop and a desktop, or between home and work computers. The working environment might consist of data that has to be copied, and of running programs in various stages of active or suspended communication with the network that have to be shut down and restarted. Why can't we just say "move this (part of the) environment to that computer" and carry on? When on a trip, why couldn't we transfer a piece of the desktop environment (for example, a forgotten open document along with its editor) to the laptop over a phone line? We would like to discover techniques to achieve all this easily and reliably.

With these motivations, we adopt a paradigm of mobility where computational ambients are hierarchically structured, where agents are confined to ambients and where ambient 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. 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, some λ-calculi, and firewall-crossing. Both Section 2 and Section 3 include an operational semantics; formal properties of the semantics are studied in the Annex.

1.1 Ambients

An ambient, in the sense in which we are going to use this word, has 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 is something that 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 in a different 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 is something that can be moved as a whole. If we reconnect a laptop to a different network, all the address spaces and file systems within it move accordingly and automatically. If we move an agent from one computer to another, its local data should move accordingly and automatically.

More precisely, we investigate ambients that have the following structure:

- Each ambient has a name. The name of an ambient is used to control access (entry, exit, communication, etc.). In a realistic situation the true name of an ambient would be guarded very closely, and only specific capabilities would be handed out about how to use the name. In our examples we are usually more liberal in the handling of names, for sake of simplicity.
- Each ambient has a collection of local agents (a.k.a. threads, processes, etc.). These are the computations that run directly within the ambient and, in a sense, control the ambient. For example, they can instruct the ambient to move.
- Each ambient has a collection of subambients. Each subambient has its own name, agents, subambients, etc.

In all of this, names are extremely important. A name is:

- something that can be created, passed around and used to name new ambients.
- something from which capabilities can be extracted.

1.2 Technical context: systems

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

- Obliq [5]. The Obliq project attacked the problems of distribution and mobility for intranet computing. It was carried out largely before the Web exploded. Within its scope, Obliq works quite well, but is not really suitable for computa-

tion and mobility over the Web, just like most other distributed paradigms developed in pre-Web days.

- Telescript [16]. Our ambient model is partially inspired by Telescript, 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.
- Java [11]. Java provides a working paradigm for mobile computation, as well as a huge amount of available and expected infrastructure on which to base more ambitious mobility efforts.
- Linda [6]. Linda is a "coordination language" where multiple processes interact in a common spaces (called a tuple space) by dropping and picking up tokens asynchronously. Distributed versions of Linda exists that use multiple tuple spaces and allow remote operations over those. 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. In particular:

- 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 are at the basis of 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 distributed 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 any other tuple space.
- 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]. This approach aims to model traditional distributed environments, along with algorithms that tolerate node failures. However, on the Internet, node failure is almost irrelevant compared with inability to reach nodes. Web servers do not often fail forever, but they frequently disappear from sight because of network or node overload, and then they come back. Sometimes they come back in a different place, for example, when a Web site changes its Internet Service Provider. Moreover, inability to reach a Web site only implies that a certain path is unavailable; it implies neither failure of that site nor global unreachability. In this sense, an observed node failure cannot simply be associated with the node itself, but instead is a property of the whole network, a property that changes over time. Our notion of locality is induced by a non-trivial and dynamic topology of locations. Failure is only represented, in a weak but realistic sense, 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 introduce communication primitives that allow us to write more natural examples.

2.1 Mobility Primitives

We first introduce a calculus in its entirety, and then we comment on the individual constructions. The syntax of the calculus is defined in the following table. The main syntactic categories are processes (including both ambients and agents that execute actions) and capabilities.

Mobility Primitives

n	names

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

Syntactic conventions

$(\nu n)P \mid Q$	is read	$((\nu n)P) \mid Q$
$!P \mid Q$	is read	$(!P) \mid Q$
$M.P \mid Q$	is read	$(M.P) \mid Q$

Abbreviations

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

The first four process primitives (restriction, inactivity, composition and replication) are commonly found in process calculi. To those we add ambients, $n[P]$, and the exercise of capabilities, $M.P$. Next we discuss these 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 term P into a new term Q.

Restriction

The restriction operator:

$$(\nu n)P$$

creates a new (unique) name n within a scope P. The new name can be used to name ambients and to operate on ambients by name.

As in the π-calculus [15], the (νn) binder can float outward as necessary to extend the scope of a name, and can float inward when possible to restrict the scope. Unlike the π-calculus, the names that are subject to scoping are not channel names, but ambient names.

The restriction construct is transparent with respect to reduction; this is expressed by the following rule:

$$P \longrightarrow Q \;\Rightarrow\; (\nu n)P \longrightarrow (\nu n)Q$$

Inaction

The process:

$$\mathbf{0}$$

is the process that does nothing. It does not reduce.

Parallel

Parallel execution is denoted by a binary operator that is commutative and associative:

$$P \mid Q$$

It obeys the rule:

$$P \longrightarrow Q \;\Rightarrow\; P \mid R \longrightarrow Q \mid R$$

This rule directly covers reduction on the left branch; reduction on the right branch is obtained by commutativity.

Replication

Replication is a technically convenient way of representing iteration and recursion. The process:

$$!P$$

denotes the unbounded replication of the process *P*. That is, $!P$ can produce as many parallel replicas of P as needed, and is equivalent to $P \mid !P$. There are no reduction rules for $!P$; in particular, the term P under ! cannot begin to reduce until it is expanded out as $P \mid !P$.

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 $n[P]$:

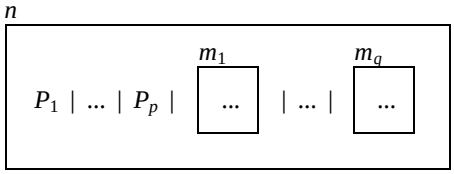
$$P \longrightarrow Q \;\Rightarrow\; n[P] \longrightarrow n[Q]$$

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) pro-

cesses 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. The general shape of an ambient is, therefore:

$$n[P_1 \mid ... \mid P_p \mid m_1[...] \mid ... \mid m_q[...]] \qquad (P_i \neq n_i[...])$$

To emphasize structure we may display ambient brackets as boxes. Then the general shape of an ambient is:



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 these capabilities is insufficient to reconstruct the original name n from which they were extracted.

Entry Capability

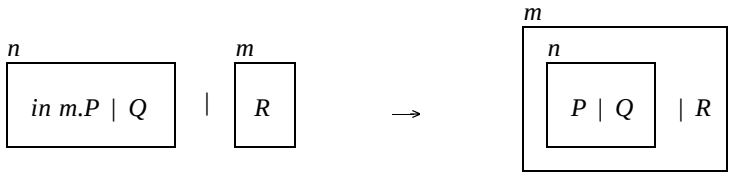
An entry capability, *in m*, can be used in the action:

$$in\ m.P$$

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

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

Or, by representing ambient brackets as boxes:



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.

Exit Capability

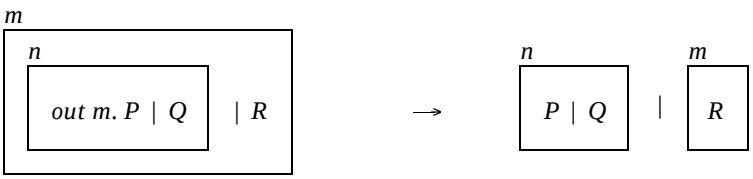
An exit capability, *out m*, can be used in the action:

$$out\ m.\ P$$

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:

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

That is:



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.

Open Capability

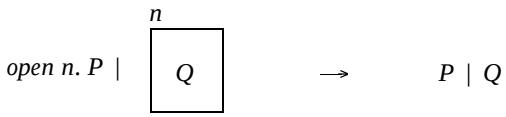
An opening capability, *open m*, can be used in the action:

$$open\ n.\ P$$

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:

$$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$$

That is:



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 (this will become clearer later in presence of communication primitives).

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, the ones we have already seen, obey the rules:

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

Objective moves (indicated by an *mv* prefix), instead obey the rules:

$$mv\ in\ m.\ P \mid m[R] \longrightarrow m[P \mid R]$$
$$m[mv\ out\ m.\ P \mid R] \longrightarrow P \mid m[R]$$

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).

Another kind of objective moves one could consider is:

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

These are objective moves that work on active ambients. However they are not as simple as the previous objective moves and, again, they can be approximated by subjective moves and *open*.

In examining these variations, one should consider who has the authority to move whom. In the case of the subjective moves, the authority rests in the top-level agents of an ambient, which naturally act as *control agents* for the ambient. In the case of objective moves, one should be careful to require enough capabilities so that ambients cannot be arbitrarily kidnapped.

Dissolution from the Inside or the Outside: Acid vs. Open

The *open* capability confers the right to dissolve an ambient and reveal its contents. As we said, this should be used carefully. There is danger both for the opening ambient, which is injected with new agents, and for the opened ambient, which loses its identity.

It is interesting to consider an operation that dissolves an ambient form the inside:

$$m[\mathit{acid}.\ P \mid Q] \longrightarrow P \mid Q$$

Acid is appealing because, for example, it gives a direct encoding of objective moves:

$$m\mathit{v}\ \mathit{in}\ n.P \triangleq (\nu q)\ q[\mathit{in}\ n.\ \mathit{acid}.\ P]$$
$$m\mathit{v}\ \mathit{out}\ n.P \triangleq (\nu q)\ q[\mathit{out}\ n.\ \mathit{acid}.\ P]$$

However, *acid* is quite dangerous. For example, if we have an entry capability for an ambient *m*, then we can entrap *m* via *acid*:

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

Therefore, *acid* should at least be regulated by an appropriate capability. We shall see that *open* can be used to define a capability-restricted version of *acid*.

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 $\longrightarrow$. We have already discussed all the reduction rules, except for one that connects reduction with equivalence. This is a semantics in the style of Milner's reaction relation [15] for the π-calculus, which was itself inspired by the Chemical Abstract Machine of Berry and Boudol [3]. Later we present an equivalent structural operational semantics.

Terms of the calculus are grouped into equivalence classes by the following relation, $\equiv$, which denotes structural congruence (that is, equivalence up to trivial syntactic restructuring). This equivalence is not a congruence; it only helps rearrange the top-level of a term where reductions can happen.

Structural Congruence

$P \equiv P$	(Struct Refl)
$P \equiv Q \ \Rightarrow\ Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \ \Rightarrow\ P \equiv R$	(Struct Trans)
$P \equiv Q \ \Rightarrow\ (\nu n)P \equiv (\nu n)Q$	(Struct Res)
$P \equiv Q \ \Rightarrow\ P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \ \Rightarrow\ !P \equiv !Q$	(Struct Repl)
$P \equiv Q \ \Rightarrow\ n[P] \equiv n[Q]$	(Struct Amb)
$P \equiv Q \ \Rightarrow\ M.P \equiv M.Q$	(Struct Action)

$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Struct Res Res)
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \quad \text{if } n \notin fn(P)$	(Struct Res Par)
$(\nu n)(m[P]) \equiv m[(\nu n)P] \quad \text{if } n \neq m$	(Struct Res Amb)
$P \mid \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Res)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Repl)

In addition, we identify terms up to renaming of bound names:

$$(\nu n)P \;=\; (\nu m)P\{n \leftarrow m\} \quad \text{if } m \notin fn(P)$$

By this we mean that these terms are understood to be identical (for example, by choosing an appropriate representation of terms), as opposed to structurally equivalent.

Note that the following terms are 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 chaining of multiple reduction steps.

Reduction

$n[in\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$	(Red Open)
$P \longrightarrow Q \;\Rightarrow\; (\nu n)P \longrightarrow (\nu n)Q$	(Red Res)
$P \longrightarrow Q \;\Rightarrow\; n[P] \longrightarrow n[Q]$	(Red Amb)
$P \longrightarrow Q \;\Rightarrow\; P \mid R \longrightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \longrightarrow Q, Q \equiv Q' \;\Rightarrow\; P' \longrightarrow Q'$	(Red $\equiv$)
$\longrightarrow^*$	reflexive and transitive closure of $\longrightarrow$

2.4 Examples

2.4.1 Locks

We can use *open* to encode locks:

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

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

$$\mathit{acquire}\ n.\ \mathit{release}\ m.\ P \mid \mathit{release}\ n.\ \mathit{acquire}\ m.\ Q$$

It appears that if we did not have *open* in the language we could not program such a handshake between two processes.

2.4.2 Firewall Access

In this example, an agent crosses a firewall by means of a previously arranged password k. We want to allow for the possibility of handling multiple passwords; therefore the name of the firewall is distinct from the password.

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, sets up a pilot ambient, $k[\mathit{in}\ k.\ \mathit{in}\ w]$, to guide agents inside. The pilot ambient enters an agent by performing $\mathit{in}\ k$ (therefore verifying that the agent knows the password), and is then given control by being opened. The execution of $\mathit{in}\ w$ then transports the agent inside the firewall, where the password wrapper is discarded.

The final effect is that the agent physically crosses into the firewall; this can be seen below by the fact that the contents Q of the agent is finally placed inside w. (For simplicity, this example is written to allow a single agent to enter.)

$$\mathit{Firewall} \triangleq (\nu w)\,(k[\mathit{in}\ k.\ \mathit{in}\ w] \mid w[\mathit{open}\ k.\ P])$$
$$\mathit{Agent} \triangleq k[\mathit{open}\ k.\ Q]$$

$$\begin{aligned}
&\mathit{Agent} \mid \mathit{Firewall}\\
&\quad \equiv (\nu w)\,(k[\mathit{open}\ k.\ Q] \mid k[\mathit{in}\ k.\ \mathit{in}\ w] \mid w[\mathit{open}\ k.\ P])\\
&\quad \rightarrow^* (\nu w)\,(k[k[\mathit{in}\ w] \mid \mathit{open}\ k.\ Q] \mid w[\mathit{open}\ k.\ P])\\
&\quad \rightarrow^* (\nu w)\,(k[\mathit{in}\ w \mid Q] \mid w[\mathit{open}\ k.\ P])\\
&\quad \rightarrow^* (\nu w)\,(w[k[Q] \mid \mathit{open}\ k.\ P])\\
&\quad \rightarrow^* (\nu w)\ w[Q \mid P]
\end{aligned}$$

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 password k.

2.4.3 Dissolution

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

$$\mathit{acid}\ n.\ P \triangleq \mathit{acid}[\mathit{out}\ n.\ \mathit{open}\ n.\ P]$$

to be used with a helper process *open acid* (an abbreviation for $\mathit{open}\ \mathit{acid}.\ \mathbf{0}$) as follows:

$$n[\mathit{acid}\ n.\ P \mid Q] \mid \mathit{open}\ \mathit{acid} \rightarrow^* P \mid Q$$

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*.

2.4.4 Objective Moves

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

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

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

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

Moreover, by picking particular names instead of *in* and *out*, ambients can restrict who can do objective moves in and out of them. These names work as keys *k*, to be used together with *allow k*:

$$\begin{array}{lll} \mathit{mv\ in}_k\ n.P & \triangleq & k[\mathit{in}\ n.\ P] \\ \mathit{mv\ out}_k\ n.P & \triangleq & k[\mathit{out}\ n.\ P] \end{array}$$

2.4.5 Synchronization on Named Channels

In CCS [13], all communication between processes is reduced to synchronization on named channels. In CCS, channels have no explicit representation other than their name. In the ambient calculus, we represent a CCS channel named *n* as follows:

$$n^{\downarrow\uparrow}[]$$

A CCS channel *n* has two complementary ports, which we shall write as *n*? and *n*!. (We use a slightly non-standard notation to avoid confusion with the notation of the ambient calculus.) These ports are conventionally thought of as input and output ports, respectively, but in fact during synchronization no value passes in either direction. Synchronization occurs between two processes attempting to synchronize on complementary ports. Process *n*?.*P* attempts to synchronize on port *n*? and then continues as *P*. Process *n*!.*P* attempts to synchronize on port *n*! and then continues as *P*. We can encode these CCS processes as follows:

$$\begin{array}{lll} n?.P & \triangleq & \mathit{mv\ in}\ n.\ \mathit{acquire\ rd.\ release\ wr.\ mv\ out}\ n.\ P \\ n!.P & \triangleq & \mathit{mv\ in}\ n.\ \mathit{release\ rd.\ acquire\ wr.\ mv\ out}\ n.\ P \end{array}$$

2.4.6 Choice

Another major feature of CCS 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 $n.P + m.Q$ reduces to P in presence of an n ambient, and reduces to Q in presence of an m ambient.

$$
\begin{array}{l}
\mathit{body\ amb\ self\ other\ R} \ \triangleq \\
\quad \mathit{in\ amb.\ out\ amb.\ in\ other.\ in\ trap.\ o[out\ trap.\ open\ o'.\ R]} \\
n.P + m.Q \ \triangleq \\
\quad (\nu\ p\ q\ trap\ o\ o') \\
\qquad (\mathit{trap}[] \mid o'[] \mid \mathit{open}\ o \mid \\
\qquad\ p[\mathit{body}\ n\ p\ q\ P \mid \mathit{open}\ q] \mid \\
\qquad\ q[\mathit{body}\ m\ q\ p\ Q \mid \mathit{open}\ p])
\end{array}
$$

We compute:

$$
\begin{array}{ll}
 & (n.P + m.Q) \mid n[R] \\
\longrightarrow^* & (\nu\ p\ q\ trap\ o\ o') \\
 & \quad (\mathit{trap}[] \mid o'[] \mid \mathit{open}\ o \mid \\
 & \quad\ n[R \mid p[\mathit{out}\ n.\ \mathit{in}\ q.\ \mathit{in\ trap.}\ o[\mathit{out\ trap.\ open}\ o'.\ P] \mid \mathit{open}\ q]] \mid \\
 & \quad\ q[\mathit{body}\ m\ q\ p\ Q \mid \mathit{open}\ p]) \\
\longrightarrow^* & (\nu\ p\ q\ trap\ o\ o') \\
 & \quad (\mathit{trap}[] \mid o'[] \mid \mathit{open}\ o \mid \\
 & \quad\ n[R] \mid p[\mathit{in}\ q.\ \mathit{in\ trap.}\ o[\mathit{out\ trap.\ open}\ o'.\ P] \mid \mathit{open}\ q] \mid \\
 & \quad\ q[\mathit{body}\ m\ q\ p\ Q \mid \mathit{open}\ p]) \\
\longrightarrow^* & (\nu\ p\ q\ trap\ o\ o') \\
 & \quad (\mathit{trap}[] \mid o'[] \mid \mathit{open}\ o \mid n[R] \mid \\
 & \quad\ q[\mathit{body}\ m\ q\ p\ Q \mid \mathit{open}\ p \mid \\
 & \quad\ p[\mathit{in\ trap.}\ o[\mathit{out\ trap.\ open}\ o'.\ P] \mid \mathit{open}\ q]]) \\
\longrightarrow^* & (\nu\ p\ q\ trap\ o\ o') \\
 & \quad (\mathit{trap}[] \mid o'[] \mid \mathit{open}\ o \mid n[R] \mid \\
 & \quad\ q[\mathit{body}\ m\ q\ p\ Q \mid \mathit{in\ trap.}\ o[\mathit{out\ trap.\ open}\ o'.\ P] \mid \mathit{open}\ q]) \\
\longrightarrow^* & (\nu\ p\ q\ trap\ o\ o') \\
 & \quad (o'[] \mid \mathit{open}\ o \mid n[R] \mid \\
 & \quad\ \mathit{trap}[q[\mathit{body}\ m\ q\ p\ Q \mid o[\mathit{out\ trap.\ open}\ o'.\ P] \mid \mathit{open}\ q]]) \\
\longrightarrow^* & (\nu\ p\ q\ trap\ o\ o') \\
 & \quad (o'[] \mid n[R] \mid \mathit{open}\ o \mid o[\mathit{open}\ o'.\ P] \mid \\
 & \quad\ \mathit{trap}[q[\mathit{body}\ m\ q\ p\ Q \mid \mathit{open}\ q]]) \\
\longrightarrow^* & (\nu\ p\ q\ trap\ o\ o') \\
 & \quad (n[R] \mid o'[] \mid \mathit{open}\ o'.\ P \mid \\
 & \quad\ \mathit{trap}[q[\mathit{body}\ m\ q\ p\ Q \mid \mathit{open}\ q]]) \\
\longrightarrow^* & n[R] \mid P \mid (\nu\ trap)\ trap[q[\mathit{body}\ m\ q\ p\ Q \mid \mathit{open}\ q]] \\
\approx & \quad n[R] \mid P \mid \mathbf{0} \\
\equiv & \quad n[R] \mid P
\end{array}
$$

We use the symbol ≈ for contextual equivalences we conjecture to hold. In the Annex, we begin to develop the necessary theory to prove these conjectured equations. In particular, by an analysis of the possible behavior of (ν *trap*) *trap*[*q*[*body m q p Q* | *open q*]] it follows that (ν *trap*) *trap*[*q*[*body m q p Q* | *open q*]] is strongly bisimilar to **0**.

2.4.7 Renaming

We can use *open* to encode a subjective ambient-renaming operation called *be*:

$$n\ be\ m.\ P \triangleq m[out\ n.\ open\ n.\ P] \mid in\ m$$

For example:

$$\begin{aligned} & n[n\ be\ m.\ P \mid Q] \equiv n[m[out\ n.\ open\ n.\ P] \mid in\ m \mid Q] \\ & \longrightarrow m[open\ n.\ P] \mid n[in\ m \mid Q] \\ & \longrightarrow m[open\ n.\ P \mid n[Q]] \\ & \longrightarrow m[P \mid Q] \end{aligned}$$

However, this operation is not atomic: a movement initiated by *Q* may disrupt it. If it is possible to plan ahead, then one can add a lock within the ambient named *n* to synchronize renaming with any movement by *Q*.

2.4.8 Seeing

We can use *open* and *be* to encode a *see* operation that detects the presence of a given ambient:

$$see\ n.\ P \triangleq (\nu\ r\ s)\ r[in\ n.\ out\ n.\ r\ be\ s.\ P] \mid open\ s$$

With this definition, *P* gets activated only if its *r* capsule can get back to the same place. That is, *P* is not activated if it is caught in the movement of *n* and ends up somewhere else.

The previous definition of *see* can detect any ambient. If an ambient wants to be seen (that is, if it contains *allow see*), then there is a simpler definition:

$$see\ n.\ P \triangleq (\nu\ seen)\ m\nu\ in_{see}\ n.\ m\nu\ out_{seen}\ n.\ P \mid open\ seen$$

2.4.9 Iteration

The following iteration construct has a number of branches $(m_i)P_i$ and a body *Q*. Each branch can be triggered by exposing an ambient $m_i[]$ in the body, which is then replaced by a copy of P_i.

$$\begin{aligned} & rec\ (m_1)P_1 \ldots (m_p)P_p\ in\ Q \triangleq \\ & \quad (\nu\ m_1 \ldots m_p)\ !open\ m_1.\ P_1 \mid \ldots \mid !open\ m_p.\ P_p \mid Q \end{aligned}$$

$$rec\ (m_1)P_1 \ldots (m_p)P_p\ in\ m_i[] \longrightarrow^*\ rec\ (m_1)P_1 \ldots (m_p)P_p\ in\ P_i$$

2.4.10 Flags

We want to indicate boolean conditions by flags in such a way that flags can be tested and execution can proceed conditionally. We assume that there are two possible flags,

flag tt for true and *flag ff* for false, and that at most one of them is present at any time. Flags are represented as follows:

$$\mathit{flag}\ n \triangleq n[]$$

A conditional operator has two branches that run in parallel, one testing for *flag tt* and then running *P*, and one testing for *flag ff* and then running *Q*. The trick is to get one branch to kill the other one when a flag is detected, so it will not accidentally activate on a future instance of the flag. To this end, when a branch detects a flag, it creates a fake flag that first traps the other branch and then activates the continuation of the first branch. We make the names *tt* and *ff* parameters of the operation:

$$\begin{array}{l}
\mathit{if}\ tt\ P, \mathit{if}\ ff\ Q \triangleq \\
\quad (\nu k)\ (k[]\ \mid \\
\qquad \mathit{open}\ tt.\ \mathit{open}\ k.\ (\nu t)\ (ff[t[]] \mid \mathit{open}\ t.\ P)\ \mid \\
\qquad \mathit{open}\ ff.\ \mathit{open}\ k.\ (\nu f)\ (tt[f[]] \mid \mathit{open}\ f.\ Q))
\end{array}$$

$$\begin{array}{l}
\mathit{flag}\ tt \quad \mid \quad \mathit{if}\ tt\ P, \mathit{if}\ ff\ Q \\
\equiv (\nu k)\ (tt[]\ \mid\ k[]\ \mid\ \mathit{open}\ tt.\ \mathit{open}\ k.\ (\nu t)\ (ff[t[]] \mid \mathit{open}\ t.\ P) \\
\qquad \mid\ \mathit{open}\ ff.\ \mathit{open}\ k.\ (\nu f)\ (tt[f[]] \mid \mathit{open}\ f.\ Q)) \\
\longrightarrow (\nu k)\ (k[]\ \mid\ \mathit{open}\ k.\ (\nu t)\ (ff[t[]] \mid \mathit{open}\ t.\ P) \\
\qquad \mid\ \mathit{open}\ ff.\ \mathit{open}\ k.\ (\nu f)\ (tt[f[]] \mid \mathit{open}\ f.\ Q)) \\
\longrightarrow (\nu k)\ (\nu t)\ (ff[t[]]\ \mid\ \mathit{open}\ t.\ P\ \mid\ \mathit{open}\ ff.\ \mathit{open}\ k.\ (\nu f)\ (tt[f[]] \mid \mathit{open}\ f.\ Q)) \\
\longrightarrow (\nu k)\ (\nu t)\ (t[]\ \mid\ \mathit{open}\ t.\ P\ \mid\ \mathit{open}\ k.\ (\nu f)\ (tt[f[]] \mid \mathit{open}\ f.\ Q)) \\
\longrightarrow P\ \mid\ (\nu k)\ \mathit{open}\ k.\ (\nu f)\ (tt[f[]] \mid \mathit{open}\ f.\ Q) \\
\approx P
\end{array}$$

$$\mathit{flip} \triangleq \mathit{if}\ tt\ ff[], \mathit{if}\ ff\ tt[]$$

The definition can be easily generalized to multiple branches. If *uu* is a third possibility, then *ff*[*t*[]] above is replaced by *ff*[*uu*[*t*[]]] and *tt*[*f*[]] is replaced by *tt*[*uu*[*f*[]]], and a third branch is added containing *tt*[*ff*[*u*[]]].

2.4.11 Numerals

We use the boolean flags of the previous section to represent numbers as ambients. For any natural number *i*, let $\underline{i}$ be the ambient-oriented numeral for *i*:

$$\begin{array}{lcl}
\underline{0} & \triangleq & num^{\downarrow\uparrow}[ff[]] \\
\underline{i+1} & \triangleq & num^{\downarrow\uparrow}[tt[]\ \mid\ \underline{i}]
\end{array}$$

These numerals are simply stacks of nested ambients, whose names alternate between *num* and *tt*, except that the inside end of the stack is the ambient $num^{\downarrow\uparrow}[ff[]]$. The number of ambients named *tt* in the stack is the number represented. For example:

$$\begin{array}{lcl}
\underline{1} & = & num^{\downarrow\uparrow}[tt[]\ \mid\ num^{\downarrow\uparrow}[ff[]]] \\
\underline{2} & = & num^{\downarrow\uparrow}[tt[]\ \mid\ num^{\downarrow\uparrow}[tt[]\ \mid\ num^{\downarrow\uparrow}[ff[]]]]
\end{array}$$

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.

$$\begin{array}{l} \mathit{ifzero}\ P\ Q \triangleq \\ \quad \mathit{mv}\ \mathit{in}\ \mathit{num}. \\ \quad\quad \mathit{if}\ \mathit{tt}\ (\mathit{tt}[] \mid \mathit{mv}\ \mathit{out}\ \mathit{num}.\ P), \\ \quad\quad \mathit{if}\ \mathit{ff}\ (\mathit{ff}[] \mid \mathit{mv}\ \mathit{out}\ \mathit{num}.\ Q) \end{array}$$

We can calculate:

$$\begin{array}{l} \underline{0} \mid \mathit{ifzero}\ P\ Q \longrightarrow^* \underline{0} \mid P \\ \underline{i{+}1} \mid \mathit{ifzero}\ P\ Q \longrightarrow^* \underline{i{+}1} \mid Q \end{array}$$

To increment or decrement a numeral, we need an auxiliary operation that runs a process as a sibling of the $\mathit{num}^{\downarrow\uparrow}[\mathit{ff}[]]$ ambient at the inside end of the stack.

$$\begin{array}{l} \mathit{inside}\ P \triangleq \\ \quad (\nu\ \mathit{it})\ \mathit{it}^{\downarrow\uparrow}[\mathit{it}[] \mid !\mathit{open}\ \mathit{it}.\ \mathit{mv}\ \mathit{out}\ \mathit{it}.\ \mathit{ifzero}\ P\ (\mathit{mv}\ \mathit{in}\ \mathit{it}.\ \mathit{in}\ \mathit{num}.\ \mathit{it}[])] \end{array}$$

We need a second auxiliary operation to signal completion from the inside end of a numeral by depositing an empty ambient named *o* at the outside end. In fact, it is easiest to deposit *o*[] as a sibling of all the ambients named *num* on the way out of the numeral.

$$\begin{array}{l} \mathit{outside}\ o \triangleq \\ \quad (\nu\ \mathit{it})\ \mathit{it}^{\downarrow\uparrow}[\mathit{it}[] \mid !\mathit{open}\ \mathit{it}.\ \mathit{mv}\ \mathit{out}\ \mathit{it}.\ (o[] \mid \mathit{mv}\ \mathit{in}\ \mathit{it}.\ \mathit{out}\ \mathit{num}.\ \mathit{it}[])] \end{array}$$

We can calculate:

$$\begin{array}{l} \underline{i} \mid \mathit{inside}\ P \approx \mathit{num}^{\downarrow\uparrow}[\mathit{tt}[] \mid \ldots\ \mathit{num}^{\downarrow\uparrow}[\mathit{tt}[] \mid P \mid \mathit{num}^{\downarrow\uparrow}[\mathit{ff}[]]]\ \ldots\ ] \\ \qquad\qquad\qquad\qquad \text{if } i \text{ is the number of } \mathit{num}\text{'s on the right-hand side} \\ (\nu o)\ (\mathit{num}^{\downarrow\uparrow}[\mathit{tt}[] \mid \ldots\ \mathit{num}^{\downarrow\uparrow}[\mathit{tt}[] \mid \mathit{outside}\ o \mid \underline{j}]\ldots] \mid \mathit{open}\ o.\ P) \approx \underline{i{+}j} \mid P \\ \qquad\qquad\qquad\qquad \text{if } o \notin \mathit{fn}(P) \text{ and } i \text{ is the number of } \mathit{num}\text{'s outside } \underline{j} \end{array}$$

Finally, we can program increment and decrement operations on numerals:

$$\begin{array}{l} \mathit{inc}\ P \triangleq \\ \quad (\nu o)\ (\mathit{inside}\ (\mathit{mv}\ \mathit{in}\ \mathit{num}.\ \mathit{open}\ \mathit{ff}.\ (\mathit{tt}[] \mid \mathit{num}^{\downarrow\uparrow}[\mathit{ff}[] \mid \mathit{outside}\ o])) \mid \mathit{open}\ o.\ P) \\ \mathit{dec}\ P \triangleq \\ \quad (\nu o)\ (\mathit{inside}\ (\mathit{open}\ \mathit{num}.\ \mathit{open}\ \mathit{tt}.\ \mathit{outside}\ o) \mid \mathit{open}\ o.\ P) \end{array}$$

These satisfy:

$$\begin{array}{l} \underline{i} \mid \mathit{inc}\ P \approx \underline{i{+}1} \mid P \\ \underline{i{+}1} \mid \mathit{dec}\ P \approx \underline{i} \mid P \end{array}$$

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

2.4.12 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:

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

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.

$$
\begin{array}{ll}
head \triangleq & \\
\quad head[!open\ S_1. & \text{state \#1 (example)} \\
\qquad mv\ out\ head. & \text{jump out to read flag} \\
\qquad\quad if\ tt\ (ff[] \mid mv\ in\ head.\ in\ sq.\ S_2[]), & \text{head right, state \#2} \\
\qquad\quad if\ ff\ (tt[] \mid mv\ in\ head.\ out\ sq.\ S_3[]) \mid & \text{head left, state \#3} \\
\qquad \ldots \mid & \text{more state transitions} \\
\qquad S_1[]] & \text{initial state}
\end{array}
$$

$$
\begin{array}{ll}
stretchRht \triangleq & \text{stretch tape right} \\
\quad (\nu r)\ r[!open\ it.\ mv\ out\ r.\ (sq^{\downarrow\uparrow}[ff[]] \mid mv\ in\ r.\ in\ sq.\ it[]) \mid it[]] &
\end{array}
$$

$$
\begin{array}{ll}
stretchLft \triangleq & \text{stretch tape left} \\
\quad !open\ it.\ mv\ in\ end. & \\
\qquad (mv\ out\ end.\ end^{\downarrow\uparrow}[sq^{\downarrow\uparrow}[] \mid ff[]] \mid & \\
\qquad\ in\ end.\ in\ sq.\ mv\ out\ end.\ open\ end.\ mv\ out\ sq.\ mv\ out\ end.\ it[]) & \\
\quad \mid it[] &
\end{array}
$$

$$
\begin{array}{ll}
machine \triangleq & \\
\quad stretchLft \mid end^{\downarrow\uparrow}[ff[] \mid head \mid stretchRht] &
\end{array}
$$

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 λ-calculus and 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 to some degree orthogonal to the mobility primitives: many such mechanisms can be added to the mobility core. However, we should try not to defeat with communication the re-

strictions 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.

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.

3.1 Communication Primitives

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

Mobility and Communication Primitives

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

New syntactic conventions

$(x).P \mid Q$ is read $((x).P) \mid Q$

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 $in\ n.\ in\ m.\ P$.

Note also that, for the purpose of communication, we have added names to the collection of capabilities. A name is a capability to create an ambient of that name.

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 mechanisms 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 only on closed terms (those not containing free variables); therefore, the definition of equivalence of section 2.3 still applies verbatim, with the understanding that P and M range now over larger classes. In addition, we have the following equivalences:

Structural Congruence

$P \equiv Q \;\Rightarrow\; M[P] \equiv M[Q]$	(Struct Amb)
$P \equiv Q \;\Rightarrow\; (x).P \equiv (x).Q$	(Struct Input)
$\varepsilon.P \equiv P$	(Struct ε)
$(M.M').P \equiv M.M'.P$	(Struct .)

We now also identify terms up to renaming of bound variables:

$$(x).P \;=\; (y).P\{x \leftarrow y\} \quad \text{if } y \notin fv(P)$$

Finally, we have a new reduction rule:

Reduction

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

3.4 Examples

3.4.1 Cells

A cell *cell c v* stores a value *v* 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*).

$$\textit{cell } c\ v \;\triangleq\; c^{\downarrow\uparrow}[\langle v \rangle \mid\ !(x).\ \langle x \rangle]$$
$$\textit{get } c\ (x).\ P \;\triangleq\; mv\ \textit{in } c.\ (x).\ (\langle x \rangle \mid\ mv\ \textit{out } c.\ P)$$
$$\textit{set } c\ \langle v \rangle.\ P \;\triangleq\; mv\ \textit{in } c.\ (x).\ (\langle v \rangle \mid\ mv\ \textit{out } c.\ P)$$

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. It is also possible to code an atomic *get-and-set* primitive, which could be used to code *test-and-set*; in that case the value expression *v* below would contain a test depending on *x*.

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

3.4.2 Records

A record is a named collection of cells. Since each cell has its own name, those names can be used as field labels:

$$\textit{record } r(l_1{=}v_1 \ldots l_n{=}v_n) \;\triangleq\; r^{\downarrow\uparrow}[\textit{cell } l_1\ v_1 \mid \ldots \mid \textit{cell } l_n\ v_n]$$
$$\textit{getr } r\ l\ (x).\ P \;\triangleq\; mv\ \textit{in } r.\ \textit{get } l\ (x).\ mv\ \textit{out } r.\ P$$
$$\textit{setr } r\ l\ \langle v \rangle.\ P \;\triangleq\; mv\ \textit{in } r.\ \textit{set } l\ \langle v \rangle.\ mv\ \textit{out } r.\ P$$

A record can contain the name of another record in one of its fields. Therefore sharing and cycles are possible.

3.4.3 Routable Packets

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*.

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

Here we assume that *P* does not interfere with routing.

3.5 Communication Between Ambients

Our basic communication primitives operate only within a given ambient. We now discuss examples of communication across ambients. In addition, in section 3.6 we treat the specific case of channel-based communication across ambients.

3.5.1 Parent I/O

We begin by considering communication between an anonymous parent and a named child (parent I/O, for short). This kind of communication is useful when a client enters a server and wants to talk to the server, as opposed to some other client who may be trying to spoof the server. The assumption is that a client can trust the server it just entered, but does not trust other clients.

Parent I/O

$^{\triangle}(x).P$	child input from parent
$^{\triangle}\langle M\rangle$	child output to parent
$^{\triangledown}n(x).P$	parent input from child n
$^{\triangledown}n\langle M\rangle$	parent output to child n

We could adopt the reduction rules:

$$^{\triangledown}n(x).P \mid n[^{\triangle}\langle M\rangle \mid Q] \longrightarrow P\{x\leftarrow M\} \mid n[Q]$$
$$^{\triangledown}n\langle M\rangle \mid n[^{\triangle}(x).P \mid Q] \longrightarrow n[P\{x\leftarrow M\} \mid Q]$$

However, it is possible to approximate parent I/O with normal ambient I/O. We assume that the names *u* and *d* (for upward and downward communication) are not used for other purposes, and we set:

$n^{\triangle}[P] \triangleq n^{\downarrow\uparrow}[u^{\downarrow\uparrow}[] \mid d^{\downarrow\uparrow}[] \mid P]$ a child $n[P]$ enabled for Parent I/O

$^{\triangle}(x).P \triangleq mv\ in\ d.\ (x).\ mv\ out\ d.\ P$

$^{\triangle}\langle M\rangle \triangleq mv\ in\ u.\ \langle M\rangle$

$^{\triangledown}n(x).P \triangleq mv\ in\ n.\ mv\ in\ u.\ (x).\ mv\ out\ u.\ mv\ out\ n.\ P$

$^{\triangledown}n\langle M\rangle \triangleq mv\ in\ n.\ mv\ in\ d.\ \langle M\rangle$

In this encoding, a child who wants to communicate with the parent deposits requests within itself, in the "mailboxes" u and d. The parent acts on those requests by first entering the child.

It is also possible to set up the parent so that it relays messages between children; if the children trust the parent, then they can also trust the origin and destination of those messages. The trust in the parent is implicit in the fact that the parent must have capabilities to enter and exit each child. These capabilities can be kept private so that children cannot interact directly.

3.5.2 Ether I/O

We now consider communication between sibling over an unnamed "ether", which is assumed to span only their enclosing ambient.

Ether I/O

$^{\approx}(x).P$	child n inputs from its parent's ether
$^{\approx}\langle M\rangle$	child n outputs to its parent's ether

We could adopt the reduction rules:

$$n[^{\approx}(x).P \mid Q] \mid m[^{\approx}\langle M\rangle \mid R] \longrightarrow n[P\{x\leftarrow M\} \mid Q] \mid m[R]$$

Again, we can emulate this kind of communication, but we must set up the parent and the children appropriately, and we must tag each I/O operation with the name of the ambient that is performing it.

$n[^{\approx}P] \triangleq n[e^{\downarrow\uparrow}[] \mid P]$ a parent $n[P]$ enabling Ether I/O

$n^{\approx}[P] \triangleq n^{\downarrow\uparrow}[P]$ a child $n[P]$ enabled for Ether I/O

$n^{\approx}(x).P \triangleq mv\ out\ n.\ mv\ in\ e.\ (x).\ mv\ out\ e.\ mv\ in\ n.\ P$

$n^{\approx}\langle M\rangle \triangleq mv\ out\ n.\ mv\ in\ e.\ \langle M\rangle$

The local "ether" is emulated by a mailbox e in the parent. This way any child can talk to any other (unspecified) child within the parent ambient.

3.5.3 Remote I/O

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 providing $!open\ io$. By M^{-1} we indicate a given return path from M.

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

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

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

To emulate Remote Procedure Call we write:

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

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

3.6 Encoding the π-calculus

One of our benchmarks of expressiveness is the ability to encode the asynchronous π-calculus. This encoding is moderately easy, given the I/O primitives. We first discuss how to represent named channels: this is the key idea for the full translation.

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.

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

These definitions satisfy the expected reduction $n(x).P \mid n\langle M\rangle \rightarrow^* P\{x\leftarrow M\}$ in presence of a channel $ch\ n$:

$$\begin{aligned}
&ch\ n \mid n(x).P \mid n\langle M\rangle \\
&\equiv (\nu p)\ (n[!open\ io] \mid io[in\ n.\ (x).\ p[out\ n.\ P]] \mid open\ p \mid io[in\ n.\ \langle M\rangle]) \\
&\rightarrow^* (\nu p)\ (n[!open\ io \mid io[(x).\ p[out\ n.\ P]] \mid io[\langle M\rangle]] \mid open\ p) \\
&\rightarrow^* (\nu p)\ (n[!open\ io \mid (x).\ p[out\ n.\ P] \mid \langle M\rangle] \mid open\ p) \\
&\rightarrow (\nu p)\ (n[!open\ io \mid p[out\ n.\ P\{x\leftarrow M\}]] \mid open\ p) \\
&\rightarrow (\nu p)\ (n[!open\ io] \mid p[P\{x\leftarrow M\}] \mid open\ p) \\
&\rightarrow (\nu p)\ (n[!open\ io] \mid P\{x\leftarrow M\}) \\
&\equiv ch\ n \mid P\{x\leftarrow M\}
\end{aligned}$$

Therefore, we can write the following encoding of the asynchronous π-calculus:

Encoding of the Asynchronous π-calculus

$$
\begin{array}{lcl}
\langle\!\langle (\nu n)P \rangle\!\rangle & \triangleq & (\nu n)\ (n[!open\ io] \mid \langle\!\langle P \rangle\!\rangle) \\
\langle\!\langle n(x).P \rangle\!\rangle & \triangleq & (\nu p)\ (io[in\ n.\ (x).\ p[out\ n.\ \langle\!\langle P \rangle\!\rangle]] \mid open\ p) \\
\langle\!\langle n\langle m \rangle \rangle\!\rangle & \triangleq & io[in\ n.\ \langle m \rangle] \\
\langle\!\langle P \mid Q \rangle\!\rangle & \triangleq & \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle \\
\langle\!\langle !P \rangle\!\rangle & \triangleq & !\langle\!\langle P \rangle\!\rangle
\end{array}
$$

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.

3.7 Encoding the λ-calculus

Since the λ-calculus can be encoded in the asynchronous π-calculus [4], and the asynchronous π-calculus can be encoded in the ambient calculus, we can in a way already encode the λ-calculus. However, we are looking here for a more direct solution.

The λ-calculus is relatively difficult to encode in the ambient calculus because processes are not values; any representation of λ-abstractions as ambients, or processes, must confront this difficulty.

In the π-calculus this issue is solved by referring to λ-abstractions via names that can be passed around as values. An operation on such a name becomes an operation on the corresponding λ-abstraction. This works because the π-calculus is "flat": all interactions occur in what we would call a single ambient. Therefore, it is always possible to find the λ-abstraction corresponding to a given name.

Instead, in our framework it is natural to introduce ambient nesting of the form *app*[*P* | *Q*] to isolate the interaction of an abstraction *P* with its argument *Q*. This nesting then gets in the way of passing names around that refer to λ-abstractions. The names can be passed around, but the λ-abstractions may happen to be at the wrong level, with respect to the names, when they need to be found.

Therefore, we use the following technique. Each occurrence of a λ-variable *x* is represented by an ambient *x*[]. A substitution is represented, to a first approximation, by !*open x*. *P*. This has the effect of replacing every occurrence of *x*[] at the current level with a copy of *P*. In addition, the substitution must be pushed across all the *app* nesting levels. Therefore, we need a substitution operator that satisfies the following recursive equation; such an operator is defined at the end of this section.

$$subst\ x\ P \quad \approx \quad (!\ open\ x.\ P) \mid (m v\ in\ app.\ subst\ x\ P)$$

The *subst* operator "floods" a term. More efficient substitutions could be obtained by preprocessing the λ-term to add information able to steer the substitutions in the right directions.

For weak reduction, we set the coding of a λ-abstraction to input a fresh name and then dissolve a surrounding *app* ambient. Application outputs a corresponding name that is replaced by a substitution operator with copies of an appropriate argument. For this encoding the variables of the λ-calculus are encoded as ambient calculus variables.

Encoding of the λ-calculus (with weak call-by-name reductions)

$$
\begin{aligned}
&\langle\!\langle x \rangle\!\rangle \triangleq x[] \\
&\langle\!\langle \lambda(x)\, b \rangle\!\rangle \triangleq (x).\, (\langle\!\langle b \rangle\!\rangle \mid \mathit{acid}\ \mathit{app}) \\
&\langle\!\langle a\ b \rangle\!\rangle \triangleq (\nu n)\ \mathit{app}^{\downarrow\uparrow}[\langle\!\langle a \rangle\!\rangle \mid \langle n \rangle \mid \mathit{subst}\ n\ \langle\!\langle b \rangle\!\rangle] \qquad n \text{ not free in } a\ b
\end{aligned}
$$

Here is a derivation of beta reduction in a context containing *allow in* | *allow out*. It assumes a suitable substitution lemma:

$$
\begin{aligned}
&\langle\!\langle (\lambda(x)\, b)(a) \rangle\!\rangle \mid \mathit{allow\ in} \mid \mathit{allow\ out} \\
\equiv\ &(\nu n)\ \mathit{app}^{\downarrow\uparrow}[(x).\, (\langle\!\langle b \rangle\!\rangle \mid \mathit{acid}\ \mathit{app}) \mid \langle n \rangle \mid \mathit{subst}\ n\ \langle\!\langle a \rangle\!\rangle] \mid \mathit{allow\ in} \mid \mathit{allow\ out} \\
\rightarrow\ &(\nu n)\ \mathit{app}^{\downarrow\uparrow}[\langle\!\langle b \rangle\!\rangle\{x \leftarrow n\} \mid \mathit{acid}\ \mathit{app} \mid \mathit{subst}\ n\ \langle\!\langle a \rangle\!\rangle] \mid \mathit{allow\ in} \mid \mathit{allow\ out} \\
\rightarrow^*\ &(\nu n)\ (\langle\!\langle b \rangle\!\rangle\{x \leftarrow n\} \mid \mathit{subst}\ n\ \langle\!\langle a \rangle\!\rangle \mid \mathit{allow\ in}) \mid \mathit{allow\ in} \mid \mathit{allow\ out} \\
\approx\ &\langle\!\langle b\{x \leftarrow a\} \rangle\!\rangle \mid \mathit{allow\ in} \mid \mathit{allow\ out}
\end{aligned}
$$

For strong reduction, we let the coding of a λ-abstraction generate a fresh name and output the name into the environment. The name is picked up by a substitution operator that proceeds to replace occurrences of the name with copies of an appropriate argument. For this encoding the variables of the λ-calculus are encoded as ambient calculus names.

Encoding of the λ-calculus (with strong reductions)

$$
\begin{aligned}
&\langle\!\langle x \rangle\!\rangle \triangleq x[] \\
&\langle\!\langle \lambda(x)\, b \rangle\!\rangle \triangleq (\nu x)\ (\langle x \rangle \mid \langle\!\langle b \rangle\!\rangle) \\
&\langle\!\langle a\ b \rangle\!\rangle \triangleq \mathit{app}^{\downarrow\uparrow}[\langle\!\langle a \rangle\!\rangle \mid (y).\, (\mathit{subst}\ y\ \langle\!\langle b \rangle\!\rangle \mid \mathit{acid}\ \mathit{app})] \qquad y \text{ not free in } b
\end{aligned}
$$

Here is a derivation of beta reduction in a context containing *allow in* | *allow out*. It assumes a suitable substitution lemma:

$$
\begin{aligned}
&\langle\!\langle (\lambda(x)\, b)(a) \rangle\!\rangle \mid \mathit{allow\ in} \mid \mathit{allow\ out} \\
\equiv\ &\mathit{app}^{\downarrow\uparrow}[((\nu x)\ \langle x \rangle \mid \langle\!\langle b \rangle\!\rangle) \mid (x).\, (\mathit{subst}\ x\ \langle\!\langle a \rangle\!\rangle \mid \mathit{acid}\ \mathit{app})] \mid \mathit{allow\ in} \mid \mathit{allow\ out} \\
\rightarrow^*\ &\mathit{app}^{\downarrow\uparrow}[((\nu x)\ \langle\!\langle b \rangle\!\rangle \mid \mathit{subst}\ x\ \langle\!\langle a \rangle\!\rangle) \mid \mathit{acid}\ \mathit{app}] \mid \mathit{allow\ in} \mid \mathit{allow\ out} \\
\rightarrow^*\ &((\nu x)\ \langle\!\langle b \rangle\!\rangle \mid \mathit{subst}\ x\ \langle\!\langle a \rangle\!\rangle) \mid \mathit{allow\ in} \mid \mathit{allow\ out} \mid \mathit{allow\ out} \\
\approx\ &\langle\!\langle b\{x \leftarrow a\} \rangle\!\rangle \mid \mathit{allow\ in} \mid \mathit{allow\ out}
\end{aligned}
$$

Note that substitution propagates inside λ-abstractions and that it is lazily executed in parallel with computation steps.

We now describe a coding of *subst x P*. First we define *flood P*, which recursively adds *P* to each node of a stack of ambients, where each ambient in the stack is named *app*. We assume that the name *it* does not occur in *P*:

$$flood\ P \triangleq (\nu\ it)\ it^{\downarrow\uparrow}[it[] \mid !open\ it.\ mv\ out\ it.\ (P \mid mv\ in\ it.\ in\ app.\ it[])]$$

We ought to be able to show:

$$flood\ P \approx P \mid mv\ in\ app.\ flood\ P$$

We define:

$$subst\ x\ P \triangleq flood\ (!open\ x.\ P)$$

and then the desired recursive equation for *subst* is satisfied.

4 Conclusions

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.

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

Acknowledgments

Thanks to Cedric Fournet and Paul McJones for comments on early drafts.

References

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

[2] Amadio, R.M., **An asynchronous model of locality, failure, and process mobility**. Rapport Interne LIM and INRIA Research Report 3109, February 1997. (To appear in 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.** *Technical Report 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**. *Communications of the ACM*, **32**(4), 444-458, 1989.

[7] Carriero, N., D. Gelernter, and L. Zuck, **Bauhaus Linda**, in *Object-Based Models and Languages for Concurrent Systems*, P. Ciancarini, O. Nierstrasz and A. Yonezawa (Ed.), Lecture Notes in Computer Science 924, Springer-Verlag, 66-76. 1995.

[8] De Nicola, R., G.-L. Ferrari and R. Pugliese, **Locality based Linda: programming with explicit localities**. *Proc. TAPSOFT'97*. (To appear). 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. 7th International Conference on Concurrency Theory (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*, Lecture Notes in Computer Science 521, 133-147, Springer Verlag, 1991.

[13] Milner, R., **A calculus of communicating systems**. Lecture Notes in Computer Science 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., **Telescript technology: the foundation for the electronic marketplace**. White Paper. General Magic, Inc. 1994.