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.
Keywords: Agents, process calculi, mobility, wide-area computation.

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, although it is being actively investigated in the coordination and agents communities.

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 lev-

* Current affiliation: Microsoft Research.

els 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 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. 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.

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* place 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 the sake of simplicity.
- Each ambient has a collection of local agents (also known as 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 became popular. Within its scope, Obliq works quite well, but is not really suitable for computation and mobility over the Web, just like most other distributed paradigms developed in pre-Web days.
- Telescript [21]. 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. Some nesting of places is allowed in Telescript.

- Java [12]. 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 [7]. Linda is a "coordination language" where multiple processes interact in a common space (called a tuple space) by dropping and picking up tokens asynchronously. Distributed versions of Linda exist that use multiple tuple spaces and allow remote operations over those. A dialect of Linda [8] 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 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 π-calculus [17] 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.
- Enrichments of the π-calculus with locations have been studied, with the aim of capturing notions of distributed computation. In the simplest form, a flat space of locations is added, and operations can be indexed by the location where they are executed. Riely and Hennessy [19] and Sewell [20] propose versions of the π-calculus extended with primitives to allow computations to migrate between named locations. The emphasis in this work is on developing type systems for mobile computation based on existing type systems for the π-calculus. Riely and Hennessy's type system regulates the usage of channel names according to permissions represented by types. Sewell's type system differentiates between local and remote channels for the sake of efficient implementation of communication.
- The join-calculus [10] 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 [11] 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 significant difference from our ambients is that movement may happen directly from any active location to any other known location.

- LLinda [9] 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, 11]. 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.
- 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.

1.4 Summary of our Approach

With respect to previous work on process calculi, we can characterize the main differences in our approach as follows. In each of the following points, our emphasis is on boundaries and their effect on computation.

- The existence of separate locations is represented by a topology of boundaries. This topology induces an abstract notion of distance between locations. Locations are not uniformly accessible, and are not identified by globally unique names.
- Process mobility is represented as crossing of boundaries. In particular, process mobility is not represented as communication of processes or process names over channels.
- Security is represented as the ability or inability to cross boundaries. In particular, security is not directly represented by cryptographic primitives or access control lists.
- Interaction between processes is by shared location within a common boundary. In particular, interaction cannot happen without proper consideration of boundaries and their topology.

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

Free names

$$fn((\nu n)P) \triangleq fn(P) - \{n\}$$
$$fn(\mathbf{0}) \triangleq \emptyset$$
$$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$$
$$fn(!P) \triangleq fn(P)$$
$$fn(n[P]) \triangleq \{n\} \cup fn(P)$$
$$fn(M.P) \triangleq fn(M) \cup fn(P)$$

$$fn(in\ n) \triangleq \{n\}$$
$$fn(out\ n) \triangleq \{n\}$$
$$fn(open\ n) \triangleq \{n\}$$

We write $P\{n \leftarrow m\}$ for the substitution of the name m for each free occurrence of the name n in the process P. Similarly for $M\{n \leftarrow m\}$.

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

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

The first four process primitives (restriction, inactivity, composition and replication) are commonly found in process calculi. To these 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 process P into a new process 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 [17], 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.

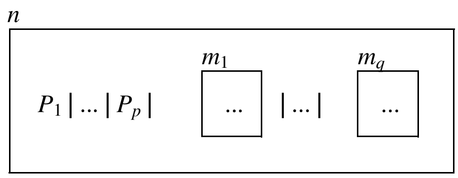
In $n[P]$, it is understood that P is actively running, and that P can be the parallel composition of several processes. We emphasize that P is running even when the surrounding ambient is moving. Running while moving may or may not be realistic, depending on the nature of the ambient and of the communication 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) 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. The general shape of an ambient is, therefore:

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

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. For each kind of capability M we have a specific rule for reducing $M.\,P$. These rules 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 m, the capability *in m* allows entry into m, the capability *out m* allows exit out of m and the capability *open m* allows the opening of m. Implicitly, the possession of one or all of these capabilities is insufficient to reconstruct the original name m from which they were extracted.

Entry Capability

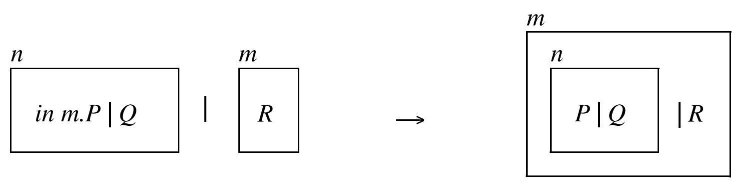
An entry capability, *in m*, can be used in the action:

$in\ m.\,P$

which instructs the ambient surrounding $in\ m.\,P$ to enter a sibling ambient named 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:

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

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

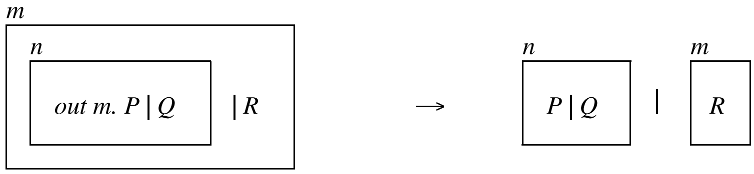
An exit capability, *out m*, can be used in the action:

$out\ m.\,P$

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 m. P

This action provides a way of dissolving the boundary of an ambient named m located at the same level as *open*, according to the rule:

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

That is:

If no ambient m can be found, the operation blocks until a time when such an ambient exists. If more than one ambient m 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 m*. P, so that the appearance of Q at the same level as P is not totally unexpected; (2) *open m* is a capability that is given out by m, so $m[Q]$ cannot be dissolved if it does not wish to be (this will become clearer later in the presence of communication primitives).

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 [16] for the π-calculus, which was itself inspired by the Chemical Abstract Machine of Berry and Boudol [3].

Processes 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).

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 processes 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 processes are understood to be identical (for example, by choosing an appropriate representation), 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 relation. 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 $\longrightarrow$

Morris-style contextual equivalence [18] is a standard way of saying that two processes have the same behavior: two processes are contextually equivalent if and only if whenever they are inserted inside an arbitrary enclosing process, they admit the same elementary observations.

In our setting, we formulate contextual equivalence in terms of observing the presence of top-level ambients. We say that a process P *exhibits an ambient named n*, and write $P{\downarrow}n$, just if P is a process containing a top-level ambient named n. We say that a process P *eventually exhibits an ambient named n*, and write $P{\Downarrow}n$, just if after some number of reductions, P exhibits an ambient named n. Formally, we define:

$$P{\downarrow}n \quad \triangleq \quad P \equiv (\nu\, m_1...m_i)\,(n[P'] \mid P'') \qquad \text{where } n \notin \{m_1...m_i\}$$
$$P{\Downarrow}n \quad \triangleq \quad P \longrightarrow^* Q \text{ and } Q{\downarrow}n$$

Next we define contextual equivalence in terms of the predicate $P{\Downarrow}n$. 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). Let *contextual equivalence* be the relation $P \simeq Q$ defined by:

$$P \simeq Q \quad \triangleq \quad \text{for all } n \text{ and } \boldsymbol{C}(),\ \boldsymbol{C}(P){\Downarrow}n \Leftrightarrow \boldsymbol{C}(Q){\Downarrow}n$$

Finally, we write $P \longrightarrow^*\simeq Q$ if there exists an R such that $P \longrightarrow^* R$ and $R \simeq Q$.

In the Appendix, we give a proof of the equation $(\nu n)n[P] \simeq \mathbf{0}$ if $n \notin fn(P)$, illustrating proof techniques for contextual equivalence. More advanced and convenient techniques will be presented in a forthcoming paper [13]. That equation allows us to garbage collect some inactive ambients, and is assumed in some of the examples in section 2.4.

2.4 Examples

In this section, we demonstrate some of the expressive power of the ambient calculus, and we discuss some expressiveness issues.

Locks

We can use *open* to encode locks. Let *release n. P* be a non-blocking operation that releases a lock n and continues with P. Let *acquire n. P* be a potentially blocking operation that attempts to acquire a lock n, and that continues with P if and when the lock is released. These operations can be defined as follows:

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

Given two locks n and m, two processes can "shake hands" before continuing with their execution:

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

Mobile Agent Authentication

A process at the top level of an ambient can be said to be privileged because it can directly affect the movement of the surrounding ambient and can open subambients. Suppose that such a privileged process wants to leave its *Home* ambient, then come back and be reinstated as a privileged process. The *Home* ambient cannot allow just any visitor to become privileged, for security reasons, so the original process must somehow be authenticated.

A solution is given below. The top level process creates a new name, n, to be used as a shared secret between itself and the *Home* ambient; *open n* is left in place to authenticate the process when it comes back. The process then leaves *Home* in the form of an *Agent* ambient. On its return inside *Home*, the *Agent* ambient exposes an n ambient, that is opened by *open n* to reinstate the continuation P of the original process at the top level of *Home*.

$$
\begin{array}{l}
Home[\\
\quad (\nu n)(open\ n \mid\\
\qquad Agent[out\ home.\ in\ home.\ n[out\ Agent.\ open\ Agent.\ P]])\\
\quad]
\end{array}
$$

Here is a trace of the computation:

$$
\begin{array}{l}
Home[(\nu n)(open\ n \mid Agent[out\ home.\ in\ home.\ n[out\ Agent.\ open\ Agent.\ P]])]\\
\quad \equiv\ (\nu n)\ Home[open\ n \mid Agent[out\ home.\ in\ home.\ n[out\ Agent.\ open\ Agent.\ P]]]\\
\quad \longrightarrow\ (\nu n)\ (Home[open\ n] \mid Agent[in\ home.\ n[out\ Agent.\ open\ Agent.\ P]])\\
\quad \longrightarrow\ (\nu n)\ Home[open\ n \mid Agent[n[out\ Agent.\ open\ Agent.\ P]]]\\
\quad \longrightarrow\ (\nu n)\ Home[open\ n \mid n[open\ Agent.\ P] \mid Agent[]]\\
\quad \longrightarrow\ (\nu n)\ Home[\mathbf{0} \mid open\ Agent.\ P \mid Agent[]]\\
\quad \longrightarrow\ (\nu n)\ Home[\mathbf{0} \mid P \mid \mathbf{0}]\\
\quad \equiv\ Home[P]
\end{array}
$$

This example illustrates the creation of a shared secret (n) within a safe location, (*Home*), the distribution of the secret over the network (carried along by *Agent*), and the authentication of incoming processes based on the shared secret.

Firewall Access

This is another example of a mobile agent trying to gain access to an ambient. In this case, though, we assume that the ambient, a firewall, keeps its name completely secret, thereby requiring authentication prior to entry.

The 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 $w \notin fn(Q)$:

$$Firewall \triangleq (\nu w)\ w[k[out\ w.\ in\ k'.\ in\ w] \mid open\ k'.\ open\ k''.\ P]$$
$$Agent \triangleq k'[open\ k.\ k''[Q]]$$

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

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 passwords.

We use an equation to express the security property of the firewall. 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 may try to disrupt it. The assumption that an attacker does not already know the password is represented by the restricted scoping of k, k', k''.

This equation is proven using techniques presented in a further paper [13].

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

One may consider alternative primitives to the ones we have adopted in the ambient calculus. In particular, 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), can be defined by the rules:

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

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 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:

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

This as an argument against taking this form of objective moves as primitive.

Dissolution

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 from 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.

However, *open* can be used to define a capability-restricted version of *acid* that does not lead to entrapment. This is a form of planned dissolution:

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

to be used with a helper process *open acid* (an abbreviation for *open acid*. **0**) 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 an exit and an open capability for *n*.

Objective Moves

Objective moves are not directly encodable. However, specific ambients can explicitly allow objective moves. Here we assume that *enter* and *exit* are two distinguished names, chosen by convention:

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

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

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

Moreover, by picking particular names instead of *enter* and *exit*, 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:

$$mv\ in_k\ n.P \triangleq k[in\ n.\ P]$$
$$mv\ out_k\ n.P \triangleq k[out\ n.\ P]$$

Synchronization on Named Channels

In CCS [15], 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:

$$n?.P \triangleq mv\ in\ n.\ acquire\ rd.\ release\ wr.\ mv\ out\ n.\ P$$
$$n!.P \triangleq mv\ in\ n.\ release\ rd.\ acquire\ wr.\ mv\ out\ n.\ P$$

Choice

A major feature of CCS [15] 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.

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

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

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

The use of $\simeq$ in this property, required for removing inert ambients, is justified by the equation discussed in Appendix.

From choice we can derive boolean conditionals. A boolean is represented by one of two flags: *flag tt* for true and *flag ff* for false. (We assume that at most one of them is present at any time.) Boolean flags and conditionals are represented as follows:

$$\begin{aligned} &\mathit{flag}\ n \ \triangleq\ n[] \\ &\mathit{if}\ \mathit{tt}\ P,\ \mathit{if}\ \mathit{ff}\ Q \ \triangleq\ \mathit{tt} \Rightarrow \mathit{open}\ \mathit{tt}.\ P + \mathit{ff} \Rightarrow \mathit{open}\ \mathit{ff}.\ Q \end{aligned}$$

Note that a boolean flag is consumed every time a branch is taken.

Renaming

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

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

For example:

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

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.

Seeing

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

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

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:

$$\mathit{see}\ n.\ P \ \triangleq\ (\nu\ \mathit{seen})\ (\mathit{mv}\ \mathit{in}_{\mathit{see}}\ n.\ \mathit{mv}\ \mathit{out}_{\mathit{seen}}\ n.\ P \mid \mathit{open}\ \mathit{seen})$$

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.

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

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

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.

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

Next, we can encode increment and decrement operations.

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

The *incsucc* operation increments a non-zero numeral $\underline{i}$. It does so by inserting an operator at the top-level of $\underline{i}$ that moves $\underline{i}$ into a further layer of *succ* ambients, thus producing $\underline{i{+}1}$. Much of the complexity of the definition is due to activating the continuation P only after the increment.

These definitions satisfy:

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

The use of $\simeq$ in the statement of these properties derives from the use of $\simeq$ in the properties of choice, and was previously discussed.

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

Turing Machines

We emulate Turing machines in a direct “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, *stretchLft* and *stretchRht*, are placed at the beginning and end of the tape and continuously add squares. If the head reaches one end of the tape and attempts to proceed further, it remains blocked until the tape has been stretched.

$head \triangleq$	
$head[!open\ S_1.$	state #1 (example)
$\quad mv\ out\ head.$	jump out to read flag
$\quad\quad if\ tt\ (ff[] \mid mv\ in\ head.\ in\ sq.\ S_2[]),$	head right, state #2
$\quad\quad if\ ff\ (tt[] \mid mv\ in\ head.\ out\ sq.\ S_3[]) \mid$	head left, state #3
$\quad ... \mid$	more state transitions
$\quad S_1[]]$	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

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

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

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 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 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.

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

Free names (revisions and additions)

$fn(M[P]) \triangleq fn(M) \cup fn(P)$
$fn((x).P) \triangleq fn(P)$
$fn(\langle M\rangle) \triangleq fn(M)$
$fn(x) \triangleq \varnothing$
$fn(n) \triangleq \{n\}$
$fn(\varepsilon) \triangleq \varnothing$
$fn(M.M') \triangleq fn(M) \cup fn(M')$

Free variables

$fv((\nu n)P) \triangleq fv(P)$
$fv(\mathbf{0}) \triangleq \varnothing$
$fv(P \mid Q) \triangleq fv(P) \cup fv(Q)$
$fv(!P) \triangleq fv(P)$
$fv(M[P]) \triangleq fv(M) \cup fv(P)$
$fv(M.P) \triangleq fv(M) \cup fv(P)$
$fv((x).P) \triangleq fv(P) - \{x\}$
$fv(\langle M\rangle) \triangleq fv(M)$
$fv(x) \triangleq \{x\}$
$fv(n) \triangleq \varnothing$
$fv(in\ M) \triangleq fv(M)$
$fv(out\ M) \triangleq fv(M)$
$fv(open\ M) \triangleq fv(M)$
$fv(\varepsilon) \triangleq \varnothing$
$fv(M.M') \triangleq fv(M) \cup fv(M')$

We write $P\{x\leftarrow M\}$ for the substitution of the capability M for each free occurrence of the variable x in the process P. Similarly for $M\{x\leftarrow M'\}$.

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 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.

A Syntactic Anomaly

To allow both names and capabilities to be output and input, there is a single syntactic sort that includes both. Hence, a meaningless term of the form $n.\ P$ can 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 type system distinguishing names from movement capabilities can avoid this anomaly [6].

3.3 Operational Semantics

For the extended calculus, 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

$P \equiv Q \;\Rightarrow\; M[P] \equiv M[Q]$	(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 processes 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)

Now that processes may contain input-bound variables, we need to modify our definition of contextual equivalence as follows: let $P \simeq Q$ if and only if for all n and for all $\boldsymbol{C}()$ such that $fv(\boldsymbol{C}(P)) = fv(\boldsymbol{C}(Q)) = \emptyset$, $\boldsymbol{C}(P){\Downarrow}n \Leftrightarrow \boldsymbol{C}(Q){\Downarrow}n$.

3.4 Examples

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.

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

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

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

Records

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

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

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

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.

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

Here we assume that P does not interfere with routing.

Remote I/O

Our basic communication primitives operate only within a given ambient. We now show examples of communication between ambients. In addition, in section 3.5 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 providing $!open\ io$. By M^{-1} we indicate a given return path from M.

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

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

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

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

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

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

3.5 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 [10] 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.

$buf\ n$	$\triangleq$	$n[!open\ io]$	a channel buffer
$(ch\ n)P$	$\triangleq$	$(\nu n)\ (buf\ n \mid P)$	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 $n(x).P \mid n\langle M\rangle \longrightarrow^* P\{x\leftarrow M\}$ in the presence of a channel buffer $buf\ n$:

$$\begin{array}{l}
buf\ n \mid n(x).P \mid n\langle M\rangle \\
\equiv\ (\nu p)\ (n[!open\ io] \mid io[in\ n.\ (x).\ p[out\ n.\ P]] \mid open\ p \mid io[in\ n.\ \langle M\rangle]) \\
\longrightarrow^*\ (\nu p)\ (n[!open\ io \mid io[(x).\ p[out\ n.\ P]] \mid io[\langle M\rangle]] \mid open\ p) \\
\longrightarrow^*\ (\nu p)\ (n[!open\ io \mid (x).\ p[out\ n.\ P] \mid \langle M\rangle] \mid open\ p) \\
\longrightarrow\ (\nu p)\ (n[!open\ io \mid p[out\ n.\ P\{x\leftarrow M\}]] \mid open\ p) \\
\longrightarrow\ (\nu p)\ (n[!open\ io] \mid p[P\{x\leftarrow M\}] \mid open\ p) \\
\longrightarrow\ (\nu p)\ (n[!open\ io] \mid P\{x\leftarrow M\}) \\
\equiv\ buf\ n \mid P\{x\leftarrow M\}
\end{array}$$

We can fairly conveniently use the above definitions of channels 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.

To make the idea of this translation precise, we fix a formalization of the asynchronous π-calculus given by the following tables. We consider a formulation where names n bound by restriction are distinct from variables x bound by input prefix. We have separate functions *fn* and *fv* for free names and free variables respectively.

The Asynchronous π-calculus

$P,Q ::=$	processes
$(\nu n)P$	restriction
$P \mid Q$	composition
$!P$	replication
$M(x).P$	input action
$M\langle M'\rangle$	async output action

$M ::=$	expressions
x	variable
n	name

Free Names and Free Variables

$fn((\nu n)P) \triangleq fn(P) - \{n\}$	$fv((\nu n)P) \triangleq fv(P)$
$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$	$fv(P \mid Q) \triangleq fv(P) \cup fv(Q)$
$fn(!P) \triangleq fn(P)$	$fv(!P) \triangleq fv(P)$
$fn(M(x).P) \triangleq fn(M) \cup fn(P)$	$fv(M(x).P) \triangleq fv(M) \cup (fv(P) - \{x\})$
$fn(M\langle M'\rangle) \triangleq fn(M) \cup fn(M')$	$fv(M\langle M'\rangle) \triangleq fv(M) \cup fv(M')$
$fn(x) \triangleq \varnothing$	$fv(x) \triangleq \{x\}$
$fn(n) \triangleq \{n\}$	$fv(n) \triangleq \varnothing$

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)
$P \equiv Q \;\Rightarrow\; M(x).P \equiv M(x).Q$	(Struct Input)
$P \mid Q \equiv Q \mid P$	(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)
$(\nu n)P \equiv P \quad \text{if } n \notin fn(P)$	(Struct Res *fn*)

Reduction

$n\langle m\rangle \mid n(x).P \longrightarrow P\{x{\leftarrow}M\}$	(Red Comm)
$P \longrightarrow Q \;\Rightarrow\; (\nu n)P \longrightarrow (\nu n)Q$	(Red Res)
$P \longrightarrow Q \;\Rightarrow\; P \mid R \longrightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \longrightarrow Q, Q \equiv Q' \;\Rightarrow\; P' \longrightarrow Q'$	(Red $\equiv$)

The encoding of the asynchronous π-calculus into the ambient calculus is given by the following translation. We translate each top-level process in the context of a set of names S that, in particular, can be taken to be the set of free names of the process.

Encoding of the Asynchronous π-calculus

$\langle\!\langle P \rangle\!\rangle_S$	$\triangleq$	$\langle\!\langle S \rangle\!\rangle \mid \langle\!\langle P \rangle\!\rangle$	where S is a set of names
$\langle\!\langle \{n_1, ..., n_k\} \rangle\!\rangle$	$\triangleq$	$n_1[!open\ io] \mid ... \mid n_k[!open\ io]$	
$\langle\!\langle (\nu n)P \rangle\!\rangle$	$\triangleq$	$(\nu n)\ (n[!open\ io] \mid \langle\!\langle P \rangle\!\rangle)$	
$\langle\!\langle P \mid Q \rangle\!\rangle$	$\triangleq$	$\langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle$	
$\langle\!\langle !P \rangle\!\rangle$	$\triangleq$	$!\langle\!\langle P \rangle\!\rangle$	
$\langle\!\langle M(x).P \rangle\!\rangle$	$\triangleq$	$(\nu p)\ (io[in\ M.\ (x).\ p[out\ M.\ \langle\!\langle P \rangle\!\rangle]] \mid open\ p)$	
$\langle\!\langle M\langle M' \rangle \rangle\!\rangle$	$\triangleq$	$io[in\ M.\ \langle M' \rangle]$	

This encoding includes the choice-free synchronous π-calculus, since it can itself be encoded within the asynchronous π-calculus [4, 14]. Moreover, since the λ-calculus can be encoded in the asynchronous π-calculus [4], we can indirectly encode the λ-calculus.

The encoding respects the semantics of the asynchronous π-calculus, in the sense that a reduction step in the asynchronous π-calculus can be emulated by a number of reduction steps and equivalences in the ambient calculus, as shown by the next proposition. It would be of interest to study questions of whether the translation preserves or reflects behavioral equivalences, but this would require more semantic machinery than we have developed in this paper.

We assume that *io* is not a name of the π-calculus.

Lemma (Substitution)

$\langle\!\langle P \rangle\!\rangle\{x \leftarrow m\} = \langle\!\langle P\{x \leftarrow m\} \rangle\!\rangle$

□

Proposition

(1) If $P \equiv P'$ holds in π and S is a set of names, then $\langle\!\langle P \rangle\!\rangle_S \simeq \langle\!\langle P' \rangle\!\rangle_S$.

(2) If $P \longrightarrow P'$ holds in π and $S \supseteq fn(P)$, then $\langle\!\langle P \rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle P' \rangle\!\rangle_S$.

Proof

Throughout this proof, we use basic properties of $\simeq$, such as the fact that it is an equivalence, a congruence, and that it includes $\equiv$ (Proposition 5-3). The places where $\simeq$ is used critically are the cases for (Struct Res *fn*) and (Red $\equiv$).

(1) We show by induction on the length of the derivation of $P \equiv P'$ that $\langle\!\langle P \rangle\!\rangle \simeq \langle\!\langle P' \rangle\!\rangle$. Then, $\langle\!\langle P \rangle\!\rangle_S \simeq \langle\!\langle P' \rangle\!\rangle_S$ (that is, $\langle\!\langle S \rangle\!\rangle \mid \langle\!\langle P \rangle\!\rangle \simeq \langle\!\langle S \rangle\!\rangle \mid \langle\!\langle P' \rangle\!\rangle$) follows by congruence of $\simeq$.

(Struct Refl), (Struct Symm), (Struct Trans), (Struct Par), (Struct Repl), (Struct Par Comm), (Struct Par Assoc), (Struct Repl Par).

Directly from the definitions and induction hypotheses.

(Struct Res) $Q \equiv Q' \Rightarrow (\nu n)Q \equiv (\nu n)Q'$.

By induction hypothesis, $\langle\!\langle Q \rangle\!\rangle \simeq \langle\!\langle Q' \rangle\!\rangle$. Since $\simeq$ is a congruence, we obtain that $(\nu n)(n[!open\ io] \mid \langle\!\langle Q \rangle\!\rangle) \simeq (\nu n)(n[!open\ io] \mid \langle\!\langle Q' \rangle\!\rangle)$. That is, $\langle\!\langle (\nu n)Q \rangle\!\rangle \simeq \langle\!\langle (\nu n)Q' \rangle\!\rangle$.

(Struct Res Res) $(\nu n)(\nu m)Q \equiv (\nu m)(\nu n)Q$.

$\langle\!\langle(\nu n)(\nu m)Q\rangle\!\rangle = (\nu n)(n[!open\ io] \mid (\nu m)(m[!open\ io] \mid \langle\!\langle Q\rangle\!\rangle)) \equiv (\nu m)(m[!open\ io] \mid (\nu n)(n[!open\ io] \mid \langle\!\langle Q\rangle\!\rangle)) = \langle\!\langle(\nu m)(\nu n)Q\rangle\!\rangle$.

(Struct Input) $Q \equiv Q' \Rightarrow M(x).Q \equiv M(x).Q'$.

By induction hypothesis, $\langle\!\langle Q\rangle\!\rangle \simeq \langle\!\langle Q'\rangle\!\rangle$. By the congruence property of $\simeq$, we have $(\nu p)\ (io[in\ M.\ (x).\ p[out\ M.\ \langle\!\langle Q\rangle\!\rangle]] \mid open\ p) \simeq (\nu p)\ (io[in\ M.\ (x).\ p[out\ M.\ \langle\!\langle Q'\rangle\!\rangle]] \mid open\ p)$. That is, $\langle\!\langle M(x).Q\rangle\!\rangle \simeq \langle\!\langle M(x).Q'\rangle\!\rangle$.

(Struct Res Par) $(\nu n)(Q' \mid Q'') \equiv Q' \mid (\nu n)Q''$ if $n \notin fn(Q')$.

Note that either $fn(\langle\!\langle Q'\rangle\!\rangle) = fn(Q')$ or $fn(\langle\!\langle Q'\rangle\!\rangle) = fn(Q') \cup \{io\}$, and that $n \neq io$ by global convention. Therefore, $n \notin fn(Q')$ implies $n \notin fn(\langle\!\langle Q'\rangle\!\rangle)$. Then, $\langle\!\langle(\nu n)(Q' \mid Q'')\rangle\!\rangle = (\nu n)(n[!open\ io] \mid (\langle\!\langle Q'\rangle\!\rangle \mid \langle\!\langle Q''\rangle\!\rangle)) \equiv \langle\!\langle Q'\rangle\!\rangle \mid (\nu n)(n[!open\ io] \mid \langle\!\langle Q''\rangle\!\rangle) = \langle\!\langle Q' \mid (\nu n)Q''\rangle\!\rangle$.

(Struct Res *fn*) $(\nu n)Q \equiv Q$ if $n \notin fn(Q)$.

As in the previous case, $n \notin fn(\langle\!\langle Q\rangle\!\rangle)$. Then, $\langle\!\langle(\nu n)Q\rangle\!\rangle = (\nu n)(n[!open\ io] \mid \langle\!\langle Q\rangle\!\rangle) \equiv \langle\!\langle Q\rangle\!\rangle \mid (\nu n)n[!open\ io]$. By Theorem 5-12, we have that $(\nu n)n[!open\ io] \simeq \mathbf{0}$. Therefore, $\langle\!\langle(\nu n)Q\rangle\!\rangle \simeq \langle\!\langle Q\rangle\!\rangle$.

(2) By induction on the length of the derivation of $P \longrightarrow P'$.

(Red Comm) $n\langle m\rangle \mid n(x).Q \longrightarrow Q\{x\leftarrow m\}$.

We need to show that if $S \supseteq fn(n\langle m\rangle \mid n(x).Q)$, then $\langle\!\langle n\langle m\rangle \mid n(x).Q\rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle Q\{x\leftarrow m\}\rangle\!\rangle_S$. We have $\langle\!\langle n\langle m\rangle \mid n(x).Q\rangle\!\rangle_S = \langle\!\langle S\rangle\!\rangle \mid io[in\ n.\ \langle m\rangle] \mid (\nu p)\ (io[in\ n.\ (x).\ p[out\ n.\ \langle\!\langle Q\rangle\!\rangle]] \mid open\ p)$.

By assumption, S includes n, and therefore $\langle\!\langle S\rangle\!\rangle$ includes $n[!open\ io]$. Then, by the computation shown at the beginning of this section and by reflexivity of $\simeq$, we obtain $\langle\!\langle n\langle m\rangle \mid n(x).Q\rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle S\rangle\!\rangle \mid \langle\!\langle Q\rangle\!\rangle\{x\leftarrow m\}$. By the substitution lemma above, the right-hand side is equal to $\langle\!\langle S\rangle\!\rangle \mid \langle\!\langle Q\{x\leftarrow m\}\rangle\!\rangle$, which is equal to $\langle\!\langle Q\{x\leftarrow m\}\rangle\!\rangle_S$.

(Red Res) $Q \longrightarrow Q' \Rightarrow (\nu n)Q \longrightarrow (\nu n)Q'$.

We need to show that if $S \supseteq fn((\nu n)Q)$, then $\langle\!\langle(\nu n)Q\rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle(\nu n)Q'\rangle\!\rangle_S$.

If $S \supseteq fn((\nu n)Q)$, then $S\cup\{n\} \supseteq fn(Q)$. Since we identify terms up to renaming of bound variables, we can assume that $n\notin S$.

By induction hypothesis, $\langle\!\langle Q\rangle\!\rangle_{S\cup\{n\}} \simeq\longrightarrow^*\simeq \langle\!\langle Q'\rangle\!\rangle_{S\cup\{n\}}$.

By repeated uses of (Red Res) and congruence of $\simeq$, we derive that $(\nu n)\langle\!\langle Q\rangle\!\rangle_{S\cup\{n\}} \simeq\longrightarrow^*\simeq (\nu n)\langle\!\langle Q'\rangle\!\rangle_{S\cup\{n\}}$. Since $(\nu n)\langle\!\langle Q\rangle\!\rangle_{S\cup\{n\}} = (\nu n)(\langle\!\langle S\cup\{n\}\rangle\!\rangle \mid \langle\!\langle Q\rangle\!\rangle) \equiv (\nu n)(n[!open\ io] \mid \langle\!\langle S\rangle\!\rangle \mid \langle\!\langle Q\rangle\!\rangle) \equiv \langle\!\langle S\rangle\!\rangle \mid \langle\!\langle(\nu n)Q\rangle\!\rangle = \langle\!\langle(\nu n)Q\rangle\!\rangle_S$, and similarly $(\nu n)\langle\!\langle Q'\rangle\!\rangle_{S\cup\{n\}} \equiv \langle\!\langle(\nu n)Q'\rangle\!\rangle_S$, we obtain that $\langle\!\langle(\nu n)Q\rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle(\nu n)Q'\rangle\!\rangle_S$.

(Red Par) $Q \longrightarrow Q' \Rightarrow Q \mid R \longrightarrow Q' \mid R$.

By induction hypothesis $\langle\!\langle Q\rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle Q'\rangle\!\rangle_S$. By repeated uses of (Red Par) and congruence of $\simeq$, we have $\langle\!\langle Q\rangle\!\rangle_S \mid \langle\!\langle R\rangle\!\rangle \simeq\longrightarrow^*\simeq \langle\!\langle Q'\rangle\!\rangle_S \mid \langle\!\langle R\rangle\!\rangle$, that is $\langle\!\langle Q \mid R\rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle Q' \mid R\rangle\!\rangle_S$.

(Red $\equiv$) $Q' \equiv Q,\ Q \longrightarrow R,\ R \equiv R' \Rightarrow Q' \longrightarrow R'$.

By induction hypothesis and (1): $\langle\!\langle Q'\rangle\!\rangle_S \simeq \langle\!\langle Q\rangle\!\rangle_S$, $\langle\!\langle Q\rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle R\rangle\!\rangle_S$, $\langle\!\langle R\rangle\!\rangle_S \simeq \langle\!\langle R'\rangle\!\rangle_S$. By transitivity of $\simeq$ we have $\langle\!\langle Q'\rangle\!\rangle_S \simeq\longrightarrow^*\simeq \langle\!\langle R'\rangle\!\rangle_S$.

□

As a corollary, we obtain that $P \longrightarrow P'$ implies $\langle\!\langle P \rangle\!\rangle_{fn(P)} \simeq\longrightarrow^*\simeq \langle\!\langle P' \rangle\!\rangle_{fn(P)}$.

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.

5 Appendix

In this appendix we assemble enough tools to prove the equation $(\nu n)n[P] \simeq \mathbf{0}$, where $n \notin fn(P)$.

We begin with some basic facts about structural congruence:

5-1 Lemma

If $P \equiv Q$ and $Q{\downarrow}n$ then $P{\downarrow}n$.

Proof

If $Q{\downarrow}n$ then $Q \equiv (\nu m_1, ..., m_k)(n[Q'] \mid Q'')$ with $n \notin \{m_1, ..., m_k\}$. By transitivity, $P \equiv (\nu m_1, ..., m_k)(n[Q'] \mid Q'')$ and therefore $P{\downarrow}n$.

□

5-2 Lemma

If $P \equiv Q$ and $Q{\Downarrow}n$ then $P{\Downarrow}n$.

Proof

By definition, $Q{\Downarrow}n$ implies $Q \longrightarrow^* R$ and $R{\downarrow}n$. Given (Red $\equiv$) and Lemma 5-1, it follows that $P \longrightarrow^* R$, and therefore that $P{\Downarrow}n$.

□

5-3 Proposition

If $P \equiv Q$ then $P \simeq Q$.

Proof

Consider any context $\boldsymbol{C}$ and any name n, and suppose that $\boldsymbol{C}(P){\Downarrow}n$. We show that $\boldsymbol{C}(Q){\Downarrow}n$. By an induction on the size of $\boldsymbol{C}()$, we get that $\boldsymbol{C}(P) \equiv \boldsymbol{C}(Q)$. By Lemma 5-2, this and $\boldsymbol{C}(P){\Downarrow}n$ imply $\boldsymbol{C}(Q){\Downarrow}n$. We may symmetrically show that $\boldsymbol{C}(Q){\Downarrow}n$ implies $\boldsymbol{C}(P){\Downarrow}n$. Hence, $P \simeq Q$.

□

The next proposition asserts that any name eventually exhibited when a context is filled with $\mathbf{0}$ is also eventually exhibited when the context is filled with any process R. A formal proof may be derived using constructions developed elsewhere [13].

5-4 Proposition

If $\boldsymbol{C}(\mathbf{0}){\Downarrow}n$ then $\boldsymbol{C}(R){\Downarrow}n$ for any R.

□

We now construct a family of relations, $\boldsymbol{S}_e$, where e is a finite set of names. We intend that $P\ \boldsymbol{S}_e\ Q$ implies P and Q are almost identical, except that for any name $n \in e$, any occurrence of n in P takes the form $n[R]$, with $n \notin fn(R)$, and the corresponding position in Q is filled with $\mathbf{0}$. By showing that the relationship $P\ \boldsymbol{S}_e\ Q$ is preserved by structural congruence and reduction we can prove our main theorem.

Let e range over finite sets of names. Let $\boldsymbol{S}_e$ be the smallest relation on processes given by the following rules.

Relation $\boldsymbol{S}_e$ on Processes

($\boldsymbol{S}$ Firewall)
$$\frac{n \in e \quad fn(P) \cap e = \varnothing}{n[P]\ \boldsymbol{S}_e\ \mathbf{0}}$$

($\boldsymbol{S}$ Res) (e' is either e or $e \cup \{m\}$)
$$\frac{P\ \boldsymbol{S}_{e'}\ Q \quad m \notin e}{(\nu m)P\ \boldsymbol{S}_e\ (\nu m)P}$$

($\boldsymbol{S}$ $\mathbf{0}$)
$$\frac{}{\mathbf{0}\ \boldsymbol{S}_e\ \mathbf{0}}$$

($\boldsymbol{S}$ Par)
$$\frac{P_1\ \boldsymbol{S}_e\ Q_1 \quad P_2\ \boldsymbol{S}_e\ Q_2}{P_1 \mid P_2\ \boldsymbol{S}_e\ Q_1 \mid Q_2}$$

($\boldsymbol{S}$ Repl)
$$\frac{P\ \boldsymbol{S}_e\ Q}{!P\ \boldsymbol{S}_e\ !Q}$$

($\boldsymbol{S}$ Amb)
$$\frac{P\ \boldsymbol{S}_e\ Q \quad m \notin e}{m[P]\ \boldsymbol{S}_e\ m[Q]}$$

($\boldsymbol{S}$ Action)
$$\frac{P\ \boldsymbol{S}_e\ Q \quad fn(M) \cap e = \varnothing}{M.P\ \boldsymbol{S}_e\ M.Q}$$

($\boldsymbol{S}$ Input)
$$\frac{P\ \boldsymbol{S}_e\ Q}{(x).P\ \boldsymbol{S}_e\ (x).Q}$$

($\boldsymbol{S}$ Output)
$$\frac{fn(M) \cap e = \varnothing}{\langle M\rangle\ \boldsymbol{S}_e\ \langle M\rangle}$$

5-5 Lemma

If $P\ \boldsymbol{S}_e\ Q$ and $fn(M) \cap e = \varnothing$ then $P\{x{\leftarrow}M\}\ \boldsymbol{S}_e\ Q\{x{\leftarrow}M\}$.

Proof

By induction on the derivation of $P\ \boldsymbol{S}_e\ Q$.

□

5-6 Lemma

If $P \equiv Q$ then $fn(P) = fn(Q)$.

Proof

By induction on the derivation of $P \equiv Q$.

□

5-7 Lemma

If $P \equiv Q$ and $Q\ \boldsymbol{S}_e\ Q'$ then there is P' with $P\ \boldsymbol{S}_e\ P'$ and $P' \equiv Q'$.

Proof

By induction on the derivation of $P \equiv Q$, we prove for all P and Q that $P \equiv Q$ implies the following:

(1) If $P\ \boldsymbol{S}_e\ P'$ then there is Q' with $Q\ \boldsymbol{S}_e\ Q'$ and $P' \equiv Q'$.

(2) If $Q\ \boldsymbol{S}_e\ Q'$ then there is P' with $P\ \boldsymbol{S}_e\ P'$ and $P' \equiv Q'$.

(Struct Refl) Trivial.

(Struct Symm) Here $P \equiv Q$ derives from $Q \equiv P$. For part (1), suppose $P\ \boldsymbol{S}_e\ P'$. By induction hypothesis (2), there is Q' with $Q\ \boldsymbol{S}_e\ Q'$ and $Q' \equiv P'$, and therefore, $P' \equiv Q'$. Part (2) follows by a symmetric argument using induction hypothesis (1).

(Struct Trans) Here $P \equiv R$ and $R \equiv Q$. For part (1), suppose $P\ \boldsymbol{S}_e\ P'$. By induction hypothesis, there is R' with $R\ \boldsymbol{S}_e\ R'$ and $P' \equiv R'$. Again, by induction hypothesis, there is Q' with $Q\ \boldsymbol{S}_e\ Q'$ and $R' \equiv Q'$. By (Struct Trans), $P' \equiv Q'$. Part (2) follows by a symmetric argument.

(Struct Res) Here $P = (\nu m)P_1$, $Q = (\nu m)Q_1$ and $P_1 \equiv Q_1$. For part (1), suppose $(\nu m)P_1\ \boldsymbol{S}_e\ P'$. This can only have been derived using ($\boldsymbol{S}$ Res). Therefore $m \notin e$ and there is e', which equals either e or $e \cup \{m\}$, and P'_1 such that P'_1 such that $P' = (\nu m)P'_1$ and $P_1\ \boldsymbol{S}_{e'}\ P'_1$. By induction hypothesis, $P_1 \equiv Q_1$ implies there is Q'_1 such that $Q_1\ \boldsymbol{S}_{e'}\ Q'_1$ and $P'_1 \equiv Q'_1$. Let $Q' = (\nu m)Q'_1$. By ($\boldsymbol{S}$ Res), $Q\ \boldsymbol{S}_e\ Q'$. By (Struct Res), $P' = (\nu m)P'_1 \equiv (\nu m)Q'_1 = Q'$. Part (2) follows by a symmetric argument.

(Struct Par) Here $P = P_1 \mid R$, $Q = Q_1 \mid R$ and $P_1 \equiv Q_1$. For part (1), suppose $P_1 \mid R\ \boldsymbol{S}_e\ P'$. This can only have been derived using ($\boldsymbol{S}$ Par), with $P' = P'_1 \mid R'$, $P_1\ \boldsymbol{S}_e\ P'_1$ and $R\ \boldsymbol{S}_e\ R'$. By induction hypothesis, there is Q'_1 with $Q_1\ \boldsymbol{S}_e\ Q'_1$ and $P'_1 \equiv Q'_1$. Let $Q' = Q'_1 \mid R'$. By ($\boldsymbol{S}$ Par), $Q\ \boldsymbol{S}_e\ Q'$ By (Struct Par), $P' \equiv Q'$. Part (2) follows by a symmetric argument.

(Struct Repl) Similar to the case for (Struct Par).

(Struct Amb) Here $P = n[P_1]$, $Q = n[Q_1]$ and $P_1 \equiv Q_1$. For part (1), suppose $n[P_1]\ \boldsymbol{S}_e\ P'$. This may be derived using one of two rules:

($\boldsymbol{S}$ Firewall) Here $P' = \mathbf{0}$, $n \in e$ and $fn(P_1) \cap e = \emptyset$. Let $Q' = \mathbf{0}$, so that $P' \equiv Q'$. By Lemma 5-6, $fn(Q_1) = fn(P_1)$, so $fn(Q_1) \cap e = \emptyset$. Hence, $Q = n[Q_1]\ \boldsymbol{S}_e\ \mathbf{0} = Q'$.

($\boldsymbol{S}$ Amb) Here $P' = n[P'_1]$, $P_1\ \boldsymbol{S}_e\ P'_1$ and $m \notin e$. By induction hypothesis, there is Q'_1 with $Q_1\ \boldsymbol{S}_e\ Q'_1$ and $P'_1 \equiv Q'_1$. Let $Q' = n[Q'_1]$. By (Struct Amb), $P' \equiv Q'$. By ($\boldsymbol{S}$ Amb), $Q\ \boldsymbol{S}_e\ Q'$.

Part (2) follows by a symmetric argument.

(Struct Action) Similar to the case for (Struct Par).

(Struct Input) Similar to the case for (Struct Par).

(Struct Res Amb) Here $P = (\nu n)m[R]$, $Q = m[(\nu n)R]$ and $m \neq n$. For part (1), suppose $(\nu n)m[R]\ \boldsymbol{S}_e\ P'$. This can only be derived using ($\boldsymbol{S}$ Res), from $m[R]\ \boldsymbol{S}_{e'}\ P'_1$, $P' = (\nu n)P'_1$, $n \notin e$ and with e' equal either to e or $e \cup \{n\}$. Moreover, the judgment $m[R]\ \mathbf{S}_{e'}\ P'_1$ can be derived using one of the following rules:

(S Firewall) Here $m \in e'$, $P'_1 = \mathbf{0}$ and $m \notin fn(R)$. Let $Q' = \mathbf{0}$. By (Struct Zero Res), $P' = (\nu n)\mathbf{0} \equiv Q'$. From $m \notin fn(R)$ it follows that $m \notin fn((\nu n)R)$. Moreover, $m \in e'$ and $m \neq n$ imply $m \in e$. Therefore, by (S Firewall), $Q = m[(\nu n)R]\ \mathbf{S}_e\ \mathbf{0} = Q'$.

(S Amb) Here $m \notin e'$, $P'_1 = m[R']$ and $R\ \mathbf{S}_{e'}\ R'$. So $P' = (\nu n)m[R']$. Let $Q' = m[(\nu n)R']$. By (Struct Res Amb), $m \neq n$ implies $P' \equiv Q'$. By (S Res), $(\nu n)R\ \mathbf{S}_e\ (\nu n)R'$. From $m \notin e'$ and $m \neq n$ it follows that $m \notin e$. Hence, by (S Amb), $Q = m[(\nu n)R]\ \mathbf{S}_e\ m[(\nu n)R'] = Q'$.

For part (2), suppose $m[(\nu n)R]\ \mathbf{S}_e\ Q'$. This can be derived using one of the following rules:

(S Firewall) Here $m \in e$, $fn((\nu n)R) \cap e = \emptyset$ and $Q' = \mathbf{0}$. Let $P' = (\nu n)\mathbf{0}$. We may assume that the bound name n is not in e, so from $fn((\nu n)R) \cap e = \emptyset$ it follows that $fn(R) \cap e = \emptyset$. By (S Firewall), this and $m \in e$ imply that $m[R]\ \mathbf{S}_e\ \mathbf{0}$. By ($S$ Res), we get $(\nu n)m[R]\ \mathbf{S}_e\ (\nu n)\mathbf{0}$, that is, $P\ \mathbf{S}_e\ P'$. By (Struct Res Zero), $P' \equiv \mathbf{0}$, that is, $P' \equiv Q'$.

(S Amb) Here $m \notin e$, $(\nu n)R\ \mathbf{S}_e\ Q'_1$ and $Q' = m[Q'_1]$. The judgment $(\nu n)R\ \mathbf{S}_e\ Q'_1$ can only be derived by (S Res), from $R\ \mathbf{S}_{e'}\ R'$ with $Q'_1 = (\nu n)R'$, e' either e or $e \cup \{n\}$, and $n \notin e$. So $Q' = m[(\nu n)R']$. Let $P' = (\nu n)m[R']$. By (Struct Res Amb), $P' \equiv Q'$. Since $m \notin e$ and $m \neq n$, we get $m \notin e'$. Therefore, by (S Amb), $m[R]\ \mathbf{S}_{e'}\ m[R']$. Moreover, since $n \notin e$ we get $(\nu n)m[R]\ \mathbf{S}_e\ (\nu n)m[R']$ by (S Res). In all, we have $P\ \mathbf{S}_e\ P'$ and $P' \equiv Q'$.

(Struct Par Comm), (Struct Par Assoc), (Struct Repl Par), (Struct Res Res), (Struct Res Par), (Struct Res Res), (Struct Zero Par), (Struct Zero Res), (Struct Zero Repl), (Struct ε), (Struct .). We omit the details of the argument for these axioms. None of them mentions ambients, and so they are easy to deal with.

□

5-8 Lemma

Whenever $P\ \mathbf{S}_e\ Q$ and $P \longrightarrow P'$ there is Q' such that $P'\ \mathbf{S}_e\ Q'$ and either $Q \longrightarrow Q'$ or $Q \equiv Q'$.

Proof

By induction on the derivation of $P \longrightarrow P'$.

(Red In) In this case $P = m[in\ p.P_1 \mid P_2] \mid p[P_3]$ and $P' = p[m[P_1 \mid P_2] \mid P_3]$. Only ($S$ Par) can derive $P\ \mathbf{S}_e\ Q$, so there are R_1 and R_2 with $m[in\ p.P_1 \mid P_2]\ \mathbf{S}_e\ R_1$, $p[P_3]\ \mathbf{S}_e\ R_2$ and $Q = R_1 \mid R_2$. Either (S Firewall) or (S Amb) can derive $m[in\ p.P_1 \mid P_2]\ \mathbf{S}_e\ R_1$.

In case (S Firewall), $m \in e$, $fn(in\ p.P_1 \mid P_2) \cap e = \emptyset$ (and therefore $p \notin e$) and $R_1 = \mathbf{0}$. Since $p \notin e$, $p[P_3]\ \mathbf{S}_e\ R_2$ must be derived by (S Amb), and not by (S Firewall), so there is Q_3 such that $R_2 = p[Q_3]$ and $P_3\ \mathbf{S}_e\ Q_3$. Therefore $Q = \mathbf{0} \mid p[Q_3]$. Let $Q' = p[\mathbf{0} \mid Q_3]$ and we have $Q \equiv Q'$. By (S Firewall), $m[P_1 \mid P_2]\ \mathbf{S}_e\ \mathbf{0}$, since $m \in e$ and since we get $fn(P_1 \mid P_2) \cap e = \emptyset$ from $fn(in\ p.P_1 \mid P_2) \cap e = \emptyset$. By ($S$ Par), $m[P_1 \mid P_2]\ \mathbf{S}_e\ \mathbf{0}$ and $P_3\ \mathbf{S}_e\ Q_3$ imply $m[P_1 \mid P_2] \mid P_3\ \mathbf{S}_e\ \mathbf{0} \mid Q_3$. By ($S$ Amb), this and $p \notin e$ imply $P' = p[m[P_1 \mid P_2] \mid P_3]\ \mathbf{S}_e\ Q'$.

In case ($\boldsymbol{S}$ Amb), $m \notin e$, $R_1 = m[R_3]$ and $in\ p.P_1 \mid P_2\ \boldsymbol{S}_e\ R_3$. By ($\boldsymbol{S}$ Par) and ($\boldsymbol{S}$ Action), there are Q_1 and Q_2 such that $P_1\ \boldsymbol{S}_e\ Q_1$, $P_2\ \boldsymbol{S}_e\ Q_2$, $R_3 = in\ p.Q_1 \mid Q_2$ and $p \notin e$. The latter implies that ($\boldsymbol{S}$ Amb), but not ($\boldsymbol{S}$ Firewall), can derive $p[P_3]\ \boldsymbol{S}_e\ R_2$. Therefore there is Q_3 such that $R_2 = p[Q_3]$ and $P_3\ \boldsymbol{S}_e\ Q_3$. In summary, we have shown that $Q = n[in\ p.Q_1 \mid Q_2] \mid p[Q_3]$. Let $Q' = p[m[Q_1 \mid Q_2] \mid Q_3]$. By (Red In), $Q \longrightarrow Q'$. By ($\boldsymbol{S}$ Amb) and ($\boldsymbol{S}$ Par), $P_i\ \boldsymbol{S}_e\ Q_i$ for $i \in 1..3$ implies that $P'\ \boldsymbol{S}_e\ Q'$.

(Red I/O) In this case, $P = \langle M\rangle \mid (x).P_3 \longrightarrow P_3\{x \leftarrow M\} = P'$. Since only ($\boldsymbol{S}$ Par) can derive $P\ \boldsymbol{S}_e\ Q$, $Q = Q_1 \mid Q_2$ with $\langle M\rangle\ \boldsymbol{S}_e\ Q_1$ and $(x).P_3\ \boldsymbol{S}_e\ Q_2$. Since these two relationships may only be derived by ($\boldsymbol{S}$ Output) and ($\boldsymbol{S}$ Input), respectively, it must be that $Q_1 = \langle M\rangle$ with $fn(M) \cap e = \emptyset$, and $Q_2 = (x).Q_3$ with $P_3\ \boldsymbol{S}_e\ Q_3$. In summary, $Q = \langle M\rangle \mid (x).Q_3$. Let $Q' = Q_3\{x \leftarrow M\}$. By (Red I/O), $Q \longrightarrow Q'$. By Lemma 5-5, $P_3\ \boldsymbol{S}_e\ Q_3$ implies $P_3\{x \leftarrow M\}\ \boldsymbol{S}_e\ Q_3\{x \leftarrow M\}$, that is, $P'\ \boldsymbol{S}_e\ Q'$.

(Red Par) In this case, $P = P_1 \mid P_2$, $P_1 \longrightarrow P'_1$ and $P' = P'_1 \mid P_2$. Since only ($\boldsymbol{S}$ Par) can derive $P\ \boldsymbol{S}_e\ Q$, $Q = Q_1 \mid Q_2$ with $P_1\ \boldsymbol{S}_e\ Q_1$ and $P_2\ \boldsymbol{S}_e\ Q_2$. By induction hypothesis, $P_1\ \boldsymbol{S}_e\ Q_1$ and $P_1 \longrightarrow P'_1$ imply there is Q'_1 with $P'_1\ \boldsymbol{S}_e\ Q'_1$ and either $Q_1 \longrightarrow Q'_1$ or $Q_1 \equiv Q'_1$. Let $Q' = Q'_1 \mid Q_2$. By ($\boldsymbol{S}$ Par), $P'\ \boldsymbol{S}_e\ Q'$. If $Q_1 \longrightarrow Q'_1$, (Red Par) implies $Q \longrightarrow Q'$. If $Q_1 \equiv Q'_1$, (Struct Par) implies $Q \equiv Q'$.

(Red Amb) In this case, $P = m[P_1]$, $P_1 \longrightarrow P'_1$ and $P' = m[P'_1]$. Either ($\boldsymbol{S}$ Firewall) or ($\boldsymbol{S}$ Amb) can derive $m[P_1]\ \boldsymbol{S}_e\ Q$. In case ($\boldsymbol{S}$ Amb), the proof is similar to the proof for (Red Par). In case ($\boldsymbol{S}$ Firewall), $m \in e$, $fn(P_1) \cap e = \emptyset$ and $Q = \mathbf{0}$. Let $Q' = \mathbf{0}$. By Lemma 5-6, $P_1 \longrightarrow P'_1$ implies $fn(P'_1) \subseteq fn(P_1)$, so $fn(P'_1) \cap e = \emptyset$. By ($\boldsymbol{S}$ Firewall), $P' = m[P'_1]\ \boldsymbol{S}_e\ \mathbf{0} = Q'$ and $Q = Q'$.

(Red ≡) In this case, $P \equiv P''$, $P'' \longrightarrow P'''$ and $P''' \equiv P'$. By Lemma 5-7, $P\ \boldsymbol{S}_e\ Q$ and $P \equiv P''$ imply there is Q'' such that $Q \equiv Q''$ and $P''\ \boldsymbol{S}_e\ Q''$. By induction hypothesis, $P''\ \boldsymbol{S}_e\ Q''$ and $P'' \longrightarrow P'''$ imply there is Q''' such that $P'''\ \boldsymbol{S}_e\ Q'''$ and either $Q'' \longrightarrow Q'''$ or $Q'' \equiv Q'''$. By Lemma 5-7, $P'''\ \boldsymbol{S}_e\ Q'''$ and $P''' \equiv P'$ imply there is Q' such that $Q''' \equiv Q'$ and $P'\ \boldsymbol{S}_e\ Q'$. From $Q \equiv Q''$, either $Q'' \longrightarrow Q'''$ or $Q'' \equiv Q'''$, and $Q''' \equiv Q'$, we obtain either $Q \equiv Q'$ by (Struct Trans), or $Q \longrightarrow Q'$ by (Red ≡).

(Red Out), (Red Open), (Red Res). Omitted. Cases (Red Out) and (Red Open) have proofs similar to (Red In). Case (Red Res) has a proof similar to (Red Par).

□

5-9 Lemma

If $P{\downarrow}n$ and $P\ \boldsymbol{S}_\emptyset\ Q$ then $Q{\downarrow}n$.

Proof

By definition, $P{\downarrow}n$ implies that $P \equiv (\nu m_1, ..., m_k)(n[P'] \mid P'')$ with $n \notin \{m_1, ..., m_k\}$. By Lemma 5-7, this and $P\ \boldsymbol{S}_\emptyset\ Q$ implies there is Q_0 with $(\nu m_1, ..., m_k)(n[P'] \mid P'')\ \boldsymbol{S}_\emptyset\ Q_0$ and $Q_0 \equiv Q$. The judgment $(\nu m_1, ..., m_k)(n[P'] \mid P'')\ \boldsymbol{S}_\emptyset\ Q_0$ can only have come from k applications of the rule ($\boldsymbol{S}$ Res); therefore, $Q_0 = (\nu m_1, ..., m_k)Q_1$ with $e \subseteq \{m_1, ..., m_k\}$ and $(n[P'] \mid P'')\ \boldsymbol{S}_{e'}\ Q_1$.

The latter judgment can have come from an application of the rule (***S*** *P*ar), and therefore $Q_1 = Q_2 \mid Q''$ with $n[P'] \ \boldsymbol{S}_e \ Q_2$ and $P'' \ \boldsymbol{S}_e \ Q''$. We know that $n \notin \{m_1, ..., m_k\}$ and therefore $n \notin e$. Hence the judgment $n[P'] \ \boldsymbol{S}_e \ Q_2$ must have come from an application of (***S*** Amb) and not from (***S*** Firewall). Therefore, $Q_2 = n[Q']$ with $P' \ \boldsymbol{S}_e \ Q'$. In all, we have that $Q_0 = (\nu m_1, ..., m_k)(n[Q'] \mid Q'')$. So $Q \equiv (\nu m_1, ..., m_k)(n[Q'] \mid Q'')$, which is to say that $Q{\downarrow}n$.

□

5-10 Lemma

If $P{\Downarrow}n$ and $P \ \boldsymbol{S}_\varnothing \ Q$ then $Q{\Downarrow}n$.

Proof

By definition, $P{\Downarrow}n$ implies that $P \longrightarrow^* P'$ and $P'{\downarrow}n$. By Lemma 5-8, $P \longrightarrow^* P'$ and $P \ \boldsymbol{S}_\varnothing \ Q$ imply there is Q' with $P' \ \boldsymbol{S}_\varnothing \ Q'$ and either $Q \longrightarrow^* Q'$ or $Q \equiv Q'$. By Lemma 5-9, $P'{\downarrow}n$ and $P' \ \boldsymbol{S}_\varnothing \ Q'$ imply that $Q'{\downarrow}n$. By definition of $\Downarrow$, either $Q \longrightarrow^* Q'$ or $Q \equiv Q'$ imply that $Q{\Downarrow}n$.

□

5-11 Proposition

If $P \ \boldsymbol{S}_\varnothing \ Q$ and $\boldsymbol{C}(P){\Downarrow}n$ then $\boldsymbol{C}(Q){\Downarrow}n$.

Proof

By an induction on the size of $\boldsymbol{C}()$, we get that $\boldsymbol{C}(P) \ \boldsymbol{S}_\varnothing \ \boldsymbol{C}(Q)$. By Lemma 5-10, $\boldsymbol{C}(P){\Downarrow}n$ and $\boldsymbol{C}(P) \ \boldsymbol{S}_\varnothing \ \boldsymbol{C}(Q)$ imply $\boldsymbol{C}(Q){\Downarrow}n$.

□

Using Propositions 5-4 and 5-11 we can prove the desired equation:

5-12 Theorem

For any process P and any name $n \notin fn(P)$, $(\nu n)n[P] \simeq \mathbf{0}$.

Proof

For any context $\boldsymbol{C}()$ and any name m we show that:

$$\boldsymbol{C}((\nu n)n[P]){\Downarrow}m \quad \Leftrightarrow \quad \boldsymbol{C}(\mathbf{0}){\Downarrow}m$$

We prove the two directions separately. First, suppose that $\boldsymbol{C}((\nu n)n[P]){\Downarrow}m$. By (***S*** Firewall), $n \notin fn(P)$ implies $n[P] \ \boldsymbol{S}_{\{n\}} \ \mathbf{0}$. By (***S*** Res), this implies $(\nu n)n[P] \ \boldsymbol{S}_\varnothing \ (\nu n)\mathbf{0}$. By Proposition 5-11, $\boldsymbol{C}((\nu n)n[P]){\Downarrow}m$ implies that $\boldsymbol{C}((\nu n)\mathbf{0}){\Downarrow}m$. By (Struct Res Zero), we get $\boldsymbol{C}((\nu n)\mathbf{0}) \equiv \boldsymbol{C}(\mathbf{0})$. By Lemma 5-2, this implies that $\boldsymbol{C}(\mathbf{0}){\Downarrow}m$. Second, suppose that $\boldsymbol{C}(\mathbf{0}){\Downarrow}m$. By Proposition 5-4, $\boldsymbol{C}(\mathbf{0} \mid (\nu n)n[P]){\Downarrow}m$. Hence, $(\nu n)n[P] \simeq \mathbf{0}$.

□

The construction of $\boldsymbol{S}_\varnothing$ lets us prove other firewall equations, not derivable from Theorem 5-12. For example, we can prove that $(\nu n)(n[P] \mid n[Q]) \simeq \mathbf{0}$ if $n \notin fn(P \mid Q)$.

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. 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**. *Proc. COORDINATION 97*, Lecture Notes in Computer Science 1282, Springer Verlag. 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] Cardelli, L., and A.D. Gordon, **Types for mobile ambients**. *Proc. 26th Annual ACM Symposium on Principles of Programming Languages*, 79-92. 1999.

[7] Carriero, N. and D. Gelernter, **Linda in context**. *Communications of the ACM*, **32**(4), 444-458, 1989.

[8] 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.

[9] De Nicola, R., G.-L. Ferrari and R. Pugliese, **Locality based Linda: programming with explicit localities**. *Proc. TAPSOFT'97*. Lecture Notes in Computer Science 1214, 712-726, Springer Verlag. 1997.

[10] 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.

[11] 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.

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

[13] Gordon, A.D, and L. Cardelli, **Equational properties of mobile ambients**. Microsoft Research Technical Report MSR-TR-99-11, February 1999.

[14] 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.

[15] Milner, R., **A calculus of communicating systems**. Lecture Notes in Computer Science 92. Springer Verlag. 1980.

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

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

[18] Morris, J.H., **Lambda-calculus models of programming languages.** Ph.D. Thesis, MIT, Dec

1968.

[19] Riely, J., and M. Hennessy, **A typed language for distributed mobile processes**. *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, 378-390, 1998.

[20] Sewell, P., **Global/local subtyping and capability inference for a distributed π-calculus**. *Proc. ICALP'98*. Lecture Notes in Computer Science 1443, 695-706, Springer, 1998.

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