Mobility Types for Mobile Ambients

Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon

Abstract. An ambient is a named cluster of processes and subambients, which moves as a group. We describe type systems able to guarantee that certain ambients will remain immobile, and that certain ambients will not be dissolved by their environment.

1 Motivation

The ambient calculus [CG98] is a process calculus that focuses primarily on process mobility rather than process communication. An ambient is a named location that may contain processes and subambients, and that can move as a unit inside or outside other ambients. Processes within an ambient may cause their enclosing ambient to move, and may communicate by anonymous asynchronous messages dropped into the local ether. Moreover, processes may open subambients, meaning that they can dissolve an ambient boundary and cause the contents of that ambient to spill into the parent ambient. The ability to move and open ambients is regulated by capabilities that processes must possess by prior knowledge or acquire by communication.

In earlier work [CG99] we studied type systems for the ambient calculus that control the exchange of values during communication. Those type systems are designed to match the communication primitives of the ambient calculus, but are able to express familiar typings for processes and functions. They are therefore successful in showing that the typed ambient calculus is as expressive as typed process and function calculi. Still, those type systems say nothing about process mobility: they guarantee that communication is well-typed wherever it may happen, but do not constrain the movement of ambients.

In this paper we study type systems that control the movement of ambients through other ambients. Our general aim is to discover type systems that can be useful for constraining the mobility behavior of agents and other entities that migrate over networks. Guarantees provided by a type system for mobility could then be used for security purposes, in the sense exemplified by Java bytecode verification [LY97]. The idea of using a type system to constrain dynamic behavior is certainly not new, but this paper makes two main contributions. First, we exhibit type systems that constrain whether or not an ambient is mobile, and whether or not an ambient may be opened. Although previous

Cardelli and Gordon are at Microsoft Research. Ghelli is at Pisa University. This paper appears in the proceedings of the *International Conference on Automata, Languages, and Programming*, Prague, Czech Republic, 11–15 July 1999. The proceedings is published by Springer Verlag as a volume of the series *Lecture Notes in Computer Science*.

authors [AP94,RH98,Sew98] use syntactic constraints to determine whether or not a process or location can move, the use of typing to draw this distinction appears to be new. Second, we propose a new mobility primitive as a solution to a problem of unwanted propagation of mobility effects from mobile ambients to those intended to be immobile.

Section 2 describes our system of mobility and locking annotations. In Section 3 we discuss the mobility primitive mentioned above. Section 4 concludes and surveys related work.

2 Mobility and Locking Annotations

This section explains our basic type system for mobility, which directly extends our previous untyped and typed calculi. Although we assume in this paper some familiarity with the untyped ambient calculus [CG98], we begin by reviewing its main features by example.

The process $a[p[\mathit{out}\ a.\mathit{in}\ b.\langle M\rangle]] \mid b[\mathit{open}\ p.(x).Q]$ models the movement of a packet p, which contains a message M, from location a to location b. The process $p[\mathit{out}\ a.\mathit{in}\ b.\langle M\rangle]$ is an ambient named p that contains a single process $\mathit{out}\,a.\mathit{in}\,b.\langle M\rangle$. It is the only subambient of the ambient named a, which itself has a sibling ambient $b[\mathit{open}\,p.(x).Q]$. Terms $\mathit{out}\,a$, $\mathit{in}\,b$, and $\mathit{open}\,p$ are capabilities, which processes exercise to cause ambients to move or to be opened.

In this example, the process $\mathit{out}\ a.\mathit{in}\ b.\langle M\rangle$ exercises the capability $\mathit{out}\ a$, which causes its enclosing ambient, the one named p, to exit its own parent, the one named a, so that $p[\mathit{in}\,b.\langle M\rangle]]$ now runs in parallel with the ambients a and b. Next, the process $\mathit{in}\,b.\langle M\rangle$ causes the ambient p to enter b, so that $p[\langle M\rangle]$ becomes a subambient of b. Up to this point, the process $\mathit{open}\,p.(x).Q$ was blocked, but now $\mathit{open}\,p$ can dissolve the boundary p. Finally, the input $(x).Q$ consumes the output $\langle M\rangle$, to leave the residue $a[] \mid b[Q\{x{\leftarrow}M\}]$, where $Q\{x{\leftarrow}M\}$ is the outcome of replacing each occurrence of x in Q with the expression M.

Two additional primitives of our calculus are replication and restriction. Just as in the π-calculus [Mil91], a replication $!P$ behaves the same as an infinite array of replicas of P running in parallel, and a restriction $(\nu n)P$ means: pick a completely fresh name, call it n, then run P.

Operational Semantics We recall the syntax of the typed ambient calculus from [CG99]. This is the same syntax as the original untyped ambient calculus [CG98], except that type annotations are added to the ν and input constructs, and that input and output are polyadic. We explain the types that appear in the syntax in the next section.

Expressions and processes

$M, N ::=$	expressions	$P, Q, R ::=$	processes
n	name	$(\nu n{:}W)P$	restriction
$\mathit{in}\,M$	can enter M	$\mathbf{0}$	inactivity
$\mathit{out}\,M$	can exit M	$P \mid Q$	composition

open M	can open M	$!P$	replication
ϵ	null M	$M[P]$	ambient
$M.M'$	path	$M.P$	action
		$(x_1{:}W_1, \ldots, x_k{:}W_k).P$	input
		$\langle M_1, \ldots, M_k \rangle$	output

A structural equivalence relation $P \equiv Q$ identifies certain processes P and Q whose behavior ought always to be equivalent:

Structural congruence ($P \equiv Q$)

$$P \mid Q \equiv Q \mid P \qquad P \equiv P$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad Q \equiv P \Rightarrow P \equiv Q$$
$$!P \equiv P \mid !P \qquad P \equiv Q, Q \equiv R \Rightarrow P \equiv R$$
$$n \neq m \Rightarrow (\nu n{:}W)(\nu m{:}W')P \equiv (\nu m{:}W')(\nu n{:}W)P$$
$$n \notin \mathit{fn}(P) \Rightarrow (\nu n{:}W)(P \mid Q) \equiv P \mid (\nu n{:}W)Q \qquad P \equiv Q \Rightarrow (\nu n{:}W)P \equiv (\nu n{:}W)Q$$
$$n \neq m \Rightarrow (\nu n{:}W)m[P] \equiv m[(\nu n{:}W)P] \qquad P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$$
$$P \mid \mathbf{0} \equiv P \qquad P \equiv Q \Rightarrow !P \equiv !Q$$
$$(\nu n{:}\mathit{Amb}^Y[^Z T])\mathbf{0} \equiv \mathbf{0} \qquad P \equiv Q \Rightarrow M[P] \equiv M[Q]$$
$$!\mathbf{0} \equiv \mathbf{0} \qquad P \equiv Q \Rightarrow M.P \equiv M.Q$$
$$\epsilon.P \equiv P \qquad P \equiv Q \Rightarrow (x_1{:}W_1, \ldots, x_k{:}W_k).P \equiv (x_1{:}W_1, \ldots, x_k{:}W_k).Q$$
$$(M.M').P \equiv M.M'.P$$

We specify process behavior via a reduction relation, $P \rightarrow Q$. The rules on the left describe the effects of, respectively, *in*, *out*, *open*, and communication.

Reduction ($P \rightarrow Q$)

$$n[\mathit{in}\, m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R] \qquad P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$$
$$m[n[\mathit{out}\, m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R] \qquad P \rightarrow Q \Rightarrow (\nu n{:}W)P \rightarrow (\nu n{:}W)Q$$
$$\mathit{open}\, n.P \mid n[Q] \rightarrow P \mid Q \qquad P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$$
$$\langle M_1, \ldots, M_k \rangle \mid (x_1{:}W_1, \ldots, x_k{:}W_k).P \rightarrow P\{x_1 \leftarrow M_1, \ldots, x_k \leftarrow M_k\} \qquad P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$$

The Type System The basic type constructions from [CG99] are the ambient types $\mathit{Amb}[T]$ and the capability types $\mathit{Cap}[T]$. A type of the form $\mathit{Amb}[T]$ describes names that name ambients that allow the exchange of T information within. A type of the form $\mathit{Cap}[T]$ is used to track the opening of ambients: it describes capabilities that may cause the unleashing of T exchanges by means of opening subambients into the current one. An exchange is the interaction of an input and an output operation within the local ether of an ambient. The exchange types, T, can be either *Shh* (no exchange allowed) or a tuple type where each component describes either a name or a capability.

In this paper, we enrich these types with two attributes indicating whether an ambient can move at all, and whether it can be opened. These attributes are intended as two of the simplest properties one can imagine that are connected with mobility. (In another paper [CGG99b] we investigate more expressive and potentially more useful generalizations of these attributes.)

We first describe the locking attributes, Y. An ambient can be declared to be either locked ($^{\bullet}$) or unlocked ($^{\circ}$). Locked ambients can never be opened, while unlocked ambients can be opened via an appropriate capability. The locking attributes are attached to the $Amb[T]$ types, which now acquire the form $Amb^{Y}[T]$. This means that any ambient whose name has type $Amb^{Y}[T]$ may (or may not) be opened, and if opened may unleash T exchanges.

We next describe the mobility attributes, Z. In general, a process can produce a number of effects that may be tracked by a type system. Previously we tracked only communication effects, T. We now plan to track both mobility and communication effects by pairs of the form ${}^{Z}T$, where Z is a flag indicating that a process executes movement operations ($^{\curvearrowright}$) or does not ($^{\veebar}$), and T is as before. A process with effects ${}^{Z}T$ should be allowed to run only within a compatible ambient, whose type will therefore have the form $Amb[{}^{Z}T]$. A capability, when used, may now cause communication effects (by open) or mobility effects (by in and out), and its type will have the form $Cap[{}^{Z}T]$.

The following table describes the syntax of our types. An ambient type $Amb^{Y}[{}^{Z}T]$ describes the name of an ambient whose locking and mobility attributes are Y and Z, respectively, and which allows T exchanges.

Types

$Y ::=$	locking annotations	$Z ::=$	mobility annotations
$\bullet$	locked	$\veebar$	immobile
$\circ$	unlocked	$\curvearrowright$	mobile
$W ::=$	message types	$T ::=$	exchange types
$Amb^{Y}[{}^{Z}T]$	ambient name	Shh	no exchange
$Cap[{}^{Z}T]$	capability	$W_1 \times \cdots \times W_k$	tuple exchange

The type rules are formally described in the next tables. There are three typing judgments: the first constructs well-formed environments, the second tracks the types of messages, and the third tracks the effects of processes. The rules for *in* and *out* introduce mobility effects, and the rule for *open* requires unlocked ambients. The handling of communication effects, T, is exactly as in [CG99].

Good environment $(E \vdash \diamond)$
Good expression of type W $(E \vdash M : W)$
Process with mobility Z **exchanging** T $(E \vdash P : {}^{Z}T)$

$$\frac{}{\varnothing \vdash \diamond} \qquad \frac{E \vdash \diamond \quad n \notin dom(E)}{E, n{:}W \vdash \diamond} \qquad \frac{E', n{:}W, E'' \vdash \diamond}{E', n{:}W, E'' \vdash n : W}$$

$$\frac{E \vdash \diamond}{E \vdash \epsilon : Cap[{}^{Z}T]} \qquad \frac{E \vdash M : Cap[{}^{Z}T] \quad E \vdash M' : Cap[{}^{Z}T]}{E \vdash M.M' : Cap[{}^{Z}T]}$$

$$\frac{E \vdash n : Amb^{Y}[{}^{Z}T]}{E \vdash in\, n : Cap[{}^{\frown}T']} \qquad \frac{E \vdash n : Amb^{Y}[{}^{Z}T]}{E \vdash out\, n : Cap[{}^{\frown}T']} \qquad \frac{E \vdash n : Amb^{\circ}[{}^{Z}T]}{E \vdash open\, n : Cap[{}^{Z}T]}$$

$$\frac{E \vdash M : Cap[{}^{Z}T] \quad E \vdash P : {}^{Z}T}{E \vdash M.P : {}^{Z}T} \qquad \frac{E \vdash M : Amb^{Y}[{}^{Z}T] \quad E \vdash P : {}^{Z}T}{E \vdash M[P] : {}^{Z'}T'}$$

$$\frac{E, n{:}Amb^{Y}[{}^{Z}T] \vdash P : {}^{Z'}T'}{E \vdash (\nu n{:}Amb^{Y}[{}^{Z}T])P : {}^{Z'}T'} \qquad \frac{E \vdash \diamond}{E \vdash \mathbf{0} : {}^{Z}T} \qquad \frac{E \vdash P : {}^{Z}T \quad E \vdash Q : {}^{Z}T}{E \vdash P \mid Q : {}^{Z}T}$$

$$\frac{E \vdash P : {}^{Z}T}{E \vdash !P : {}^{Z}T} \qquad \frac{E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P : {}^{Z}W_1 \times \cdots \times W_k}{E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).P : {}^{Z}W_1 \times \cdots \times W_k}$$

$$\frac{E \vdash M_1 : W_1 \quad \cdots \quad E \vdash M_k : W_k}{E \vdash \langle M_1, \ldots, M_k \rangle : {}^{Z}W_1 \times \cdots \times W_k}$$

For example, consider the untyped process discussed at the beginning of this section. Suppose that the message M has type W. We can type the process under the assumption that a is a locked, immobile ambient ($Amb^{\bullet}[{}^{\curlyvee}Shh]$), that p is an unlocked, mobile ambient ($Amb^{\circ}[{}^{\frown}W]$), and that b is a locked, mobile ambient ($Amb^{\bullet}[{}^{\frown}W]$). More formally, under the assumptions $E \vdash a : Amb^{\bullet}[{}^{\curlyvee}Shh]$, $E \vdash p : Amb^{\circ}[{}^{\frown}W]$, $E \vdash b : Amb^{\bullet}[{}^{\frown}W]$, $E \vdash M : W$, and $E, x{:}W \vdash P : {}^{\frown}W$ we can derive that $E \vdash a[p[out\ a.in\ b.\langle M \rangle]] \mid b[open\ p.(x{:}W).P] : {}^{\curlyvee}Shh$.

As customary, we can prove a subject reduction theorem asserting the soundness of the typing rules. It can be interpreted as stating that every communication is well-typed, that no locked ambient will ever be opened, and that no *in* or *out* will ever act on an immobile ambient. As in earlier work [CG99], the proof is by induction on derivations.

Theorem 1. *If $E \vdash P : {}^{Z}T$ and $P \to Q$ then $E \vdash Q : {}^{Z}T$.*

Remark 1. The type system of [CG99] can be embedded in the current type system by taking $Amb[T] = Amb^{\circ}[{}^{\frown}T]$ and $Cap[T] = Cap[{}^{\frown}T]$.

Encoding Channels Communication in the basic ambient calculus happens in the local ether of an ambient. Messages are simply dropped into the ether, without specifying a recipient other than any process that does or will exist in the current ambient. Even within the ambient calculus, though, one often feels the need of additional communication operations, whether primitive or derived.

The familiar mechanism of communication over named channels, used by most process calculi, can be expressed fairly easily in the untyped ambient calculus. We should think, though, of a channel as a new entity that may reside within an ambient. In particular, communications executed on the same channel name but in separate ambients will not interact, at least until those ambients are somehow merged.

The basic idea for representing channels is as follows; see [CG98,CG99] for details. If c is the name of a channel we want to represent, then we use a name

c^b to name an ambient that acts as a communication buffer for c. We also use a name c^p to name ambients that act as communication packets directed at c. The buffer ambient opens all the incoming packet ambients and lets their contents interact. So, an output on channel c is represented as a c^p packet that enters c^b (where it is opened up) and that contains an output operation. Similarly, an input on channel c is represented as a c^p packet that enters c^b (where it is opened up) and that contains an input operation; after the input is performed, the rest of the process exits the buffer appropriately to continue execution. The creation of a channel name c is represented as the creation of the two names c^b and c^p. Similarly, the communication of a channel name c is represented as the communication of the two names c^b and c^p.

This encoding of channels can be typed within the type system of [CG99]. Let $Ch[T]$ denote the type of a channel c exchanging messages of type T. This type can be represented as $Amb[T] \times Amb[T]$, which is the type of the pair of names c^b, c^p. Packets named c^p have exchange type T by virtue of performing corresponding inputs and outputs. Buffers named c^p have exchange type T by virtue of opening c^p packets, and unleashing their exchanges.

The natural question now is whether we can type this encoding of channels in the type system given earlier. This can be done trivially by Remark 1, by making all the ambients movable and openable. But this solution is not very satisfactory. In particular, now that we have a type system for mobility, we would like to declare the communication buffers to be both immobile and locked, so that channel communication cannot be disrupted by accidental or malicious activities. Note, for example, that a malicious packet could contain instructions that would cause the buffer to move when the packet is opened. Such a packet should be ruled out as untypable if we made the buffer immobile.

The difficulty with protecting buffers from malicious packets does not arise if we use a systematic translation of a high-level channel abstraction into the lower-level ambient calculus. However, in a situation where code is untrusted (for example, mobile code received from the network), we cannot assume that the ambient-level code interacting with the channel buffers is the image of a high-level abstraction. Thus, we would like to typecheck the untrusted code to make sure that it satisfies the mobility constraints of the trusted environment.

We now encounter a fundamental difficulty that will haunt us for the rest of this section and all of the next. In the type system given earlier, we cannot declare buffers to be immobile, because buffers open packets that are mobile; therefore, buffers are themselves potentially mobile. Packets must, of course, be mobile because they must enter buffers.

We have explored several possible solutions to this problem. In the rest of this section we present a different (more complex) encoding of channels that satisfies several of our wishes. In the next section we add a typed primitive that allows us to use an encoding similar to the original one; this new primitive has other applications as well. In addition, one could investigate more complex type systems that attempt to capture the fact that a well-behaved packet moves once and then becomes immobile, or some suitable generalization of this notion.

The idea for the encoding shown below comes from [CG99], where an alternative encoding of channels is presented. In that encoding there are no buffers; the packets, though, are self-coalescing, so that each packet can act as an exchange buffer for another packet. Here we combine the idea of self-coalescing packets with an immobile buffer that contains them. Since nothing is opened directly within the buffer, the difficulty with constraining the mobility of buffers, described above, disappears. A trace of the difficulty, though, remains in that a process performing an input must be given a mobile type, even when it performs only channel communications.

We formalize our encoding of channels by considering a calculus obtained from the system given earlier by adding operations for creating typed channels ($(\nu c{:}Ch[W_1,\ldots,W_k])P$) and for inputs and outputs over them ($c\langle n_1,\ldots,n_k\rangle$ and $c(x_1{:}W_1,\ldots,x_k{:}W_k).P$, where c is the channel name, and the n_i's are other channel names that are communicated over it). The additional rules for typing channels are as follows:

Channel I/O, where $W = Ch[W_1,\ldots,W_k]$

$$\frac{E, n{:}Ch[T] \vdash P : {}^{Z}T'}{E \vdash (\nu n{:}Ch[T])P : {}^{Z}T'} \qquad \frac{E \vdash c : W \quad E, x_1{:}W_1,\ldots,x_k{:}W_k \vdash P : {}^{Z}T \quad Z = \curvearrowright}{E \vdash c(x_1{:}W_1,\ldots,x_k{:}W_k).P : {}^{Z}T}$$

$$\frac{E \vdash c : W \quad E \vdash M_1 : W_1 \quad \cdots \quad E \vdash_c M_k : W_k}{E \vdash c\langle M_1,\ldots,M_k\rangle : {}^{Z}T}$$

We can translate this extended calculus into the core calculus described earlier. Here, we show the translation of channel types and channel operations; the other types and operations are translated in a straightforward way. We show the complete translation in the full version of this paper [CGG99a].

Expressing channels with ambients

$$\begin{array}{lcl}
\llbracket Ch[W_1,\ldots,W_k]\rrbracket^p & = & Amb^{\circ}[{\curvearrowright}\llbracket W_1\rrbracket^b \times \llbracket W_1\rrbracket^p \ldots \times \llbracket W_k\rrbracket^b \times \llbracket W_k\rrbracket^p] \\
\llbracket Ch[W_1,\ldots,W_k]\rrbracket^b & = & Amb^{\bullet}[{\veebar}Shh] \\
\llbracket(\nu c{:}Ch[W_1,\ldots,W_k])P\rrbracket & = & (\nu c^b{:}\llbracket Ch[W_1,\ldots,W_k]\rrbracket^b) \\
 & & \quad (\nu c^p{:}\llbracket Ch[W_1,\ldots,W_k]\rrbracket^p)(c^b[\,] \mid \llbracket P\rrbracket) \\
\llbracket c\langle n_1,\ldots,n_k\rangle\rrbracket & = & c^p[in\ c^b.(!open\ c^p \mid in\ c^p \mid \langle n_1^b, n_1^p,\ldots,n_k^b,n_k^p\rangle)] \\
\llbracket c(x_1{:}W_1,\ldots,x_k{:}W_k).P_{\curvearrowright T}\rrbracket & = & (\nu s{:}Amb^{\circ}[{\curvearrowright}T])(open\ s \mid c^p[in\ c^b.(!open\ c^p \mid in\ c^p \mid \\
 & & \quad (x_1^b{:}\llbracket W_1\rrbracket^b, x_1^p{:}\llbracket W_1\rrbracket^p,\ldots,x_k^b{:}\llbracket W_k\rrbracket^b, x_k^p{:}\llbracket W_k\rrbracket^p). \\
 & & \quad s[!out\ c^p \mid out\ c^b.\llbracket P\rrbracket])])
\end{array}$$

(The translation $\llbracket c(x_1{:}W_1,\ldots,x_k{:}W_k).P_{\curvearrowright T}\rrbracket$ depends on the type of the process P, which we indicate by the subscript ${\curvearrowright}T$.) Note how a channel named c is encoded by an ambient named c^b whose type is immobile and locked. Therefore, the type system guarantees that the channel cannot be tampered with by rogue processes.

We can show that if $E \vdash P : {}^{Z}T$ is derivable in the calculus extended with channels then $\llbracket E\rrbracket \vdash \llbracket P\rrbracket : {}^{Z}\llbracket T\rrbracket$ is derivable in the original calculus, where $\llbracket E\rrbracket$,

$[\![P]\!]$, and $[\![T]\!]$ are the translations of the environments, processes and exchange types of the extended calculus. Hence, this translation demonstrates a typing of channels in which channels are immobile ambients. However, a feature of this typing is that in an input $c(x_1{:}W_1, \ldots, x_k{:}W_k).P$, the process P is obliged to be mobile. The next section provides a type system that removes this obligation.

3 Objective moves

The movement operations of the standard ambient calculus are called "subjective" because they have the flavor of "I (ambient) wish to move there". Other movement operations are called "objective" when they have the flavor of "you (ambient) should move there". Objective moves can be adequately emulated with subjective moves [CG98]: the latter were chosen as primitive on the grounds of expressive power and simplicity.

Certain objective moves, however, can acquire additional interpretations with regard to the typing of mobility. In this section we introduce objective moves, and we distinguish between subjective-mobility annotations (the ones of Section 2) and objective-mobility annotations. It is perhaps not too surprising that the introduction of typing constructs requires the introduction of new primitives. For example, in both the π-calculus and the ambient calculus, the introduction of simple types requires a switch from monadic to polyadic I/O.

We consider an objective move operation that moves to a different location an ambient that has not yet started. It has the form $go\, N.M[P]$ and has the effect of starting the ambient $M[P]$ in the location reached by following, if possible, the path N. Note that P does not become active until after the movement is completed.

Unlike *in* and *out*, this *go* operation does not move the ambient enclosing the operation. Possible interpretations of this operation are to install a piece of code at a given location and then run it, or to move the continuation of a process to a given location.

When assigning a mobility type to the *go* operation, we can now make a subtle distinction. The ambient $M[P]$ is moved, objectively, from one place to another. But after it gets there, maybe the process P never executes subjective moves, and therefore M can be declared subjectively immobile. Moreover, the *go* operation itself does not cause its surrounding ambient to move, so it may also be possible to declare the surrounding ambient subjectively immobile.

Therefore, we can move an ambient from one place to another without noticing any subjective-mobility effects. Still, something got moved, and we would like to be able to track this fact in the type system. For this purpose, we introduce objective-mobility annotations, attached to ambients that may be objectively moved. In particular, an ambient may be objectively mobile, but subjectively immobile.

In conclusion, we achieve the task, impossible in the type system of Section 2, of moving an immobile ambient, once. (More precisely, the possible encodings of the *go* operation in terms of subjective moves are not typable in the type system of Section 2 if we set M to be immobile.) The additional expressive power can be used to give a better typing to communication channels, by causing a communication packet to move into a buffer without requiring the packet to be itself mobile, and therefore without having to require the buffer that opens the packet to be mobile.

To formalize these ideas, we make the following changes to the system of Section 2. Using objective moves we can type an encoding of channels which eliminates the immobility obligation noted at the end of the previous section. Moreover, in the full paper [CGG99a], objective moves are essential for encoding an example language, in which mobile threads migrate between immobile hosts.

Additions to process syntax, structural congruence, and reduction

$P, Q, R ::= \mathit{go}\ N.M[P]$	objective move	
$\cdots$	as in Section 2	
$\mathit{go}\ \epsilon.M[P] \equiv M[P]$		$\mathit{go}\ (\mathit{in}\ m.N).n[P] \mid m[Q] \rightarrow m[\mathit{go}\ N.n[P] \mid Q]$
$P \equiv Q \Rightarrow$		$m[\mathit{go}\ (\mathit{out}\ m.N).n[P] \mid Q] \rightarrow \mathit{go}\ N.n[P] \mid m[Q]$
$\quad \mathit{go}\ N.M[P] \equiv \mathit{go}\ N.M[Q]$		

The types of the system extended with objective moves are the same as the types in Section 2, except that the types of ambient names are $\mathit{Amb}^{Y\,Z'}[{}^{Z}T]$, where Y is a locking annotation, T is an exchange type, and Z' and Z are an objective-mobility annotation and a subjective-mobility annotation, respectively.

Modifications and additions to type syntax and typing

$\mathit{Amb}^{Y}[{}^{Z}T]$ becomes $\mathit{Amb}^{Y\,Z''}[{}^{Z}T]$ in the syntax of types and in all the rules where it appears. Add the following rule:

$$\frac{E \vdash N : \mathit{Cap}[{}^{Z'}T'] \quad E \vdash M : \mathit{Amb}^{Y\,Z'}[{}^{Z}T] \quad E \vdash P : {}^{Z}T}{E \vdash \mathit{go}\ N.M[P] : {}^{Z''}T''}$$

Theorem 2. *If $E \vdash P : {}^{Z}T$ and $P \rightarrow Q$ then $E \vdash Q : {}^{Z}T$.*

4 Conclusions and Related Work

We have argued [CG98,Car99,CG99,GC99] that the idea of an ambient is a useful and general abstraction for expressing and reasoning about mobile computation. In this paper, we qualified the ambient idea by introducing type systems that distinguish between mobile and immobile, and locked and unlocked ambients. Thus qualified, ambients better describe the structure of mobile computations.

The type systems presented in this paper derive from our earlier work on exchange types for ambients [CG99]. That type system tracks the types of messages that may be input or output within each ambient; it is analogous to Milner's sort system for the π-calculus [Mil91], which tracks the types of messages that may be input or output on each channel.

Our mobility annotations govern the ways in which an ambient can be moved. The data movement types of the mobile λ-calculus of Sekiguchi and Yonezawa [SY97] also govern movement, the movement of variables referred to by mobile processes. Their data movement types are checked dynamically, rather than statically. In the setting of the π-calculus, various type systems have been proposed to track the distinction between local and remote references to channels [Ama97,Sew98,SWP98], but none of these systems tracks process mobility.

Our locking annotations allow static checking of a simple security property: that nobody will attempt to open a locked ambient. More complex type systems than ours demonstrate that more sophisticated security properties of concurrent systems can be checked statically: access control [DFPV98,HR98b], allocation of permissions [RH98], and secrecy and integrity properties [Aba97,HR98a,SV98]. Ideas from some of these systems may be applicable to ambients.

References

[Aba97] M. Abadi. Secrecy by typing in security protocols. In *Proceedings TACS'97*, *LNCS* 1281, pages 611–638. Springer, 1997.

[Ama97] R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings COORDINATION 97*, *LNCS* 1282. Springer, 1997.

[AP94] R. M. Amadio and S. Prasad. Localities and failures. In *Proceedings FST&TCS'94*, *LNCS* 880, pages 205–216. Springer, 1994.

[Car99] L. Cardelli. Abstractions for mobile computation. In *Secure Internet Programming*, *LNCS*. Springer, 1999.

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings FoSSaCS'98*, *LNCS* 1378, pages 140–155. Springer, 1998.

[CG99] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings POPL'99*, pages 79–92. ACM, January 1999.

[CGG99a] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility Types for Mobile Ambients. Technical Report, Microsoft Research, 1999 (to appear).

[CGG99b] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. Unpublished, 1999.

[DFPV98] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. Submitted for publication, 1998.

[GC99] A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. In *Proceedings FoSSaCS'99*, *LNCS* 1578, pages 212–226. Springer, 1999.

[HR98a] N. Heintz and J. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings POPL'98*, pages 365–377. ACM, 1998.

[HR98b] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *Proceedings HLCL'98*, *ENTCS* 16(3). Elsevier, 1998.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS–LFCS–91–180, University of Edinburgh, October 1991.

[RH98] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings POPL'98*, pages 378–390. ACM, 1998.

[Sew98] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings ICALP'98*, *LNCS* 1443, pages 695–706. Springer, 1998.

[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings POPL'98*, pages 355–364. ACM, 1998.

[SWP98] P. Sewell, P. T. Wojciechowski, and B.C. Pierce. Location independence for mobile agents. In *Workshop on Internet Programming Languages*, 1998.

[SY97] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In *Proceedings FMOODS'97*, pages 21–36. IFIP, 1997.