Types for the Ambient Calculus

Luca Cardelli[†] and Giorgio Ghelli[‡] and Andrew D. Gordon[†]
[†]*Microsoft Research, 1 Guildhall Street, Cambridge, UK*
[‡] *Università di Pisa, Dipartimento di Informatica, Corso Italia 40, Pisa, Italy*

The ambient calculus is a concurrent calculus where the unifying notion of 'ambient' is used to model many different constructs for distributed and mobile computation. We study a type system that describes several properties of ambient behavior. The type system allows ambients to be partitioned in disjoint sets (*groups*), according to the intended design of a system, in order to specify both the communication and the mobility behavior of ambients.

CONTENTS

1. INTRODUCTION

The ambient calculus [13] is a process calculus whose basic abstraction, the *ambient*, represents mobile, nested, computational environments, with local communications. Ambients can represent the standard components of distributed systems, such as nodes, channels, messages, and mobile code. They can also represent situations where entire active computational environments are moved, as happens with mobile computing devices, and with multithreaded mobile agents.

We define here a set of type systems for the ambient calculus, which are based on the idea of partitioning ambients in progammer defined *groups*, and tracking communication and mobility properties.

Type systems are, today, a widely applied technique allowing programmers to describe the key properties of their code, and to have these properties mechanically and efficiently checked. Mobile code makes types, and machine-checkable properties in general, useful for security reasons too, as has been demonstrated by the checking performed on Java applets [26].

In standard languages, the key invariants that are maintained by type systems have mainly to do with the contents of variables and with the interfaces of functions, procedures, or methods. In the ambient calculus, the basic properties of a piece of code are those related to its mobility, to the possibility of opening an ambient and exposing its content, and to the type of data which may be exchanged inside an ambient. To understand how groups arise in this context, consider a typical static property we may want to express in a type system for the ambient calculus; informally:

The ambient named n can enter the ambient named m.

This could be expressed as a typing $n : \mathit{CanEnter}(m)$ stating that n is a member of the collection $\mathit{CanEnter}(m)$ of names that can enter m. However, this would bring us straight into the domain of dependent types [14], since the type $\mathit{CanEnter}(m)$ depends on the name m. Instead, we introduce type-level groups of names, G, H, and restate our property as:

The name m belongs to group G.
The ambient named n can enter any ambient of group G.

This idea leads to typings of the form: $m : G$, $n : \mathit{CanEnter}(G)$ which are akin to standard typings such as $x : \mathit{Int}$, $y : \mathit{Channel}(\mathit{Int})$.

To appreciate the relevance of groups in the description of distributed systems, consider a programmer coding a typical distributed system composed of nodes and mobile threads moving from one node to another, and where threads communicate by sending input and output packets through typed channels. In this paper we define a type system where a programmer can:

• define groups such as *Node*, *Thread*, *Channel*, and *Packet*, which match the system structure;

• declare properties such as: this ambient is a *Thread* and it may only cross ambients which are *Nodes*; this ambient is a *Packet* and can enter *Channels*; this ambient is a *Channel* of type T, and it cannot move or be opened, and it may open *Packets* containing data of type T; this ambient is a *Node* and it cannot move or be opened;

• have the system statically verify all these properties.

Our *groups* are similar to *sorts* used in typed versions of the π-calculus [27], but we introduce an operation, $(\nu G)P$, for creating a new group G, which can be used within the process P.

The binders for new groups, (νG), can float outward during reduction as long as this adjustment (called extrusion in the π-calculus) does not introduce name clashes. Because of extrusion, group binders do not impede the mobility of ambients that are enclosed in the initial scope of fresh groups but later move away. On the other hand, even though extrusion enlarges scopes, simple scoping restrictions in the typing rules prevent names belonging to a fresh group from ever being received by a process which has been defined outside the initial scope of the group.

Therefore, we obtain a flexible way of protecting the propagation of names. This is to be contrasted with the situation in the untyped π-calculus and ambient calculus, where names can (intentionally, accidentally, or maliciously) be extruded arbitrarily far, by the automatic and unrestricted application of extrusion rules, and communicated to other parties.

This paper reports the results of a research effort some parts of which are described in conference papers. In [12] we investigate *exchange types*, which subsume standard type systems for processes and functions, but do not impose restrictions on mobility; no groups were present in that system. In [9] we report on *immobility* and *locking* annotations, which are basic predicates about mobility, still with no notion of groups. In [10] we introduce the notion of groups; that paper is essentially an extended abstract of the present one.

We organise the paper as follows. In Section 2 we review the basic untyped ambient calculus. In Section 3 we informally introduce a group-based exchange type system which only tracks communications. In Section 4 we give a precise definition of the same system, and a subject reduction result. Section 5 enriches the system of Section 4 to control ambient opening. In Section 6, we define the full system in which both ambient opening and ambient movement are tracked. Section 7 formalizes safety properties guaranteed by typing. In Section 8 we revisit a typed encoding of a distributed programming language from our earlier work on locking and mobility annotations [9], in order to illustrate the expressiveness of the type system. In particular, we show how groups help describing the different classes of ambients and their properties. Section 9 concludes and discusses related work. Finally, appendixes contain proofs of the subject reduction and effect safety properties for the full type system.

2. THE POLYADIC AMBIENT CALCULUS (REVIEW)

We begin by reviewing and slightly extending the ambient calculus of [13]. In that calculus, communication is based on the exchange of single values. Here we extend the calculus with communication based on tuples of values (polyadic communication), since this simple extension greatly facilitates the task of providing an expressive type system. We also add objective moves, as in [9], and we annotate bound variables with type information.

Four of our process constructions (restriction, inactivity, composition, and replication) are commonly found in process calculi. To these we add ambients, capabilities, and a simple form of communication. We briefly discuss these constructions; see [13] for a more detailed introduction.

The restriction operator, $(\nu n{:}W)P$, creates a new (unique) name n of type W within a scope P. The new name can be used to name ambients and to operate on ambients by name. The inactive process, $\mathbf{0}$, does nothing. Parallel composition is denoted by a binary operator, $P \mid Q$, that is commutative and associative. Replication is a technically convenient way of representing iteration and recursion: the process $!P$ denotes the unbounded replication of the process P and is equivalent to $P \mid !P$.

An ambient is written $M[P]$, where M is the name of the ambient, and P is the process running inside the ambient.

The process $M.P$ executes an action regulated by the capability M, and then continues as the process P. We consider three kinds of capabilities: one for entering an ambient, one for exiting an ambient, and one for opening up an ambient. (The latter requires special care in the type system.) Capabilities are obtained from names; given a name n, the capability $in\, n$ allows entry into n, the capability $out\, n$ allows exit out of n and the capability $open\, n$ allows the opening of n. Implicitly, the possession of one or all of these capabilities is insufficient to reconstruct the original name n from which they were extracted. Capabilities can also be composed into paths, $M.M'$, with ϵ for the empty path.

Communication is asynchronous and local to an ambient. It is similar to channel communication in the asynchronous π-calculus [7, 21], except that the channel has no name: the surrounding ambient provides the context where the communication happens. The process $\langle M_1, \ldots, M_k \rangle$ represents the output of a tuple of values, with no continuation. The process $(x_1{:}W_1, \ldots, x_k{:}W_k).P$ represents the input of a tuple of values, whose components are bound to $x_1, \ldots, x_k$, with continuation P.

Communication is used to exchange both names and capabilities, which share the same syntactic class M of messages. The first task of our type system is to distinguish the Ms that are names from the Ms that are capabilities, so that each is guaranteed to be used in an appropriate context. In general, the type system might distinguish other kinds of expressions, such as integer and boolean expressions, but we do not include those in our basic calculus.

The process $go\, N.M[P]$ moves the ambient $M[P]$ as specified by the N capability, and has $M[P]$ as its continuation. It is called an *objective move* since the ambient $M[P]$ is moved from the outside, while a movement caused by a process $N.P$ which runs inside an ambient is called a *subjective move*. There are more powerful forms of objective move, beyond what is expressible in the untyped calculus, that may have undesirable properties [13]. We adopt the form $go\ N.M[P]$ as primitive because it usefully allows more refined typings than are possible with only subjective moves—as we show in Section 6.2—and because it does not affect the untyped operational semantics, since it is derivable in the untyped calculus. We can define an objective move $go\ N.M[P]$ to be short for $(\nu k)k[N.M[out\ k.P]]$ where k is not free in P.

Messages and Processes:

$M, N ::=$	message
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
$P, Q, R ::=$	process
$(\nu n{:}W)P$	restriction
$\mathbf{0}$	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	action
$(x_1{:}W_1, \ldots, x_k{:}W_k).P$	input action
$\langle M_1, \ldots, M_k \rangle$	output action

$go\ N.M[P]$	objective move

The following table displays the main reduction rules of the calculus (the full set is presented in Section 4). The notation $P\{x_1{\leftarrow}M_1, \ldots, x_k{\leftarrow}M_k\}$ in rule (Red I/O) denotes the outcome of a capture-avoiding simultaneous substitution of message M_i for each free occurrence of the corresponding name x_i in the process P, for $i \in 1..k$.

Reduction:

$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.P \mid n[Q] \rightarrow P \mid Q$	(Red Open)
$\langle M_1, \ldots, M_k \rangle \mid (x_1{:}W_1, \ldots, x_k{:}W_k).P \rightarrow$ $P\{x_1{\leftarrow}M_1, \ldots, x_k{\leftarrow}M_k\}$	(Red I/O)
$go(in\ m.N).n[P] \mid m[Q] \rightarrow m[go\ N.n[P] \mid Q]$	(Red Go In)
$m[go(out\ m.N).n[P] \mid Q] \rightarrow go\ N.n[P] \mid m[Q]$	(Red Go Out)

We use the following syntactic conventions:

- parentheses may be used for precedence
- $(\nu n{:}W)P \mid Q$ is read $((\nu n{:}W)P) \mid Q$
- $!P \mid Q$ is read $(!P) \mid Q$
- $M.P \mid Q$ is read $(M.P) \mid Q$
- $M.M'.P$ is read $M.(M'.P)$
- $(n_1{:}W_1, \ldots, n_k{:}W_k).P \mid Q$ is read $((n_1{:}W_1, \ldots, n_k{:}W_k).P) \mid Q$
- $n[] \triangleq n[\mathbf{0}]$
- $M \triangleq M.\mathbf{0}$ (where appropriate)

As an example, consider the following process:

$$a[p[out\ a.in\ b.\langle c \rangle]] \mid b[open\ p.(x).x[]]$$

Intuitively, this example represents a packet named p moving from a machine a to a machine b. The process $p[out\ a.in\ b.\langle c \rangle]$ represents the packet, as a subambient of ambient a. The name of the packet ambient is p, and its interior is the process $out\ a.in\ b.\langle c \rangle$. This process consists of three sequential actions: exercise the capability $out\ a$, exercise the capability $in\ b$, and then output the name c. The effect of the two capabilities on the enclosing ambient p is to move p out of a and into b (rules (Red Out), (Red In)), to reach the state:

$$a[] \mid b[p[\langle c \rangle] \mid open\ p.(x).x[]]$$

In this state, the interior of a is empty but the interior of b consists of two running processes, the subambient $p[\langle c \rangle]$ and the process $openp.(x).x[]$. This process is attempting to exercise the $open\ p$ capability. This capability was previously blocked, but now that the p ambient is present, the capability's effect is to dissolve the ambient's boundary; hence, the interior of b becomes the process $\langle c \rangle \mid (x).x[]$

(Red Open). This is a composition of an output $\langle c\rangle$ with an input $(x).x[]$. The input consumes the output, leaving $c[]$ as the interior of b (Red I/O). Hence, the final state of the whole example is $a[] \mid b[c[]]$.

As an example for the objective moves, consider the following variation of the previous one:

$$a[go(out\ a.in\ b).p[\langle c\rangle]] \mid b[open\ p.(x).x[]]$$

In this case, the ambient $p[\langle c\rangle]$ is moved from the outside, out of a and into b (rules (Red Go Out), (Red Go In)), to reach the same state that was reached in the previous version after the (Red Out), (Red In) subjective moves:

$$a[] \mid b[go\ \epsilon.p[\langle c\rangle] \mid open\ p.(x).x[]]$$

3. INTRODUCTION TO EXCHANGE TYPES

An ambient is a place where processes can exchange messages and where other ambients can enter and exit. We introduce here a type system which regulates communication, while mobility will be tackled in the following sections. This system generalizes the one presented in [12] by allowing the partitioning of ambients into groups.

3.1. Topics of Conversation

Within an ambient, multiple processes can freely execute input and output actions. Since the messages are undirected, it is easily possible for a process to utter a message that is not appropriate for some receiver. The main idea of the exchange type system is to keep track of the topic of conversation that is permitted within a given ambient, so that talkers and listeners can be certain of exchanging appropriate messages.

The range of topics is described in the following table by message types, W, and exchange types, T. The message types are $G[T]$, the type of names of ambients which belong to the group G and that allow exchanges of type T, and $Cap[T]$, the type of capabilities that when used may cause the unleashing of T exchanges (as a consequence of opening ambients that exchange T). The exchange types are Shh, the absence of exchanges, and $W_1 \times \ldots \times W_k$, the exchange of tuples of messages with elements of the respective message types. For $k = 0$, the empty tuple type is called $\mathbf{1}$; it allows the exchange of empty tuples, that is, it allows pure synchronization. The case $k = 1$ allows any message type to be an exchange type.

Types:

$W ::=$	message type
$G[T]$	name in group G for ambients allowing T exchanges
$Cap[T]$	capability unleashing T exchanges
$S, T ::=$	exchange type
Shh	no exchange
$W_1 \times \cdots \times W_k$	tuple exchange ($\mathbf{1}$ is the null product)

For example, in a scope where the *Agent* and *Place* groups have been defined, we can express the following types:

• An ambient of the *Agent* group where no exchange is allowed (a quiet *Agent*): $Agent[Shh]$

• A harmless capability: $Cap[Shh]$

• A *Place* where names of quiet *Agent*s may be exchanged:

$$Place[Agent[Shh]]$$

• A *Place* where harmless capabilities may be exchanged:

$$Place[Cap[Shh]]$$

• A capability that may unleash the exchange of names of quiet *Agent*s:

$$Cap[Agent[Shh]]$$

3.2. Intuitions

Before presenting the formal type rules (in Section 4), we discuss the intuitions that lead to them.

3.2.1. Typing of Processes

If a message M has message type W, then $\langle M \rangle$ is a process that outputs (exchanges) W messages. Therefore, we have a rule stating that:

$$M : W \quad \text{implies} \quad \langle M \rangle : W$$

If P is a process that may exchange W messages, then $(x{:}W).P$ is also a process that may exchange W messages. Therefore:

$$P : W \quad \text{implies} \quad (x{:}W).P : W$$

The process $\mathbf{0}$ exchanges nothing, so it naturally has exchange type *Shh*. However, we may also consider $\mathbf{0}$ as a process that may exchange any type. This is useful when we need to place $\mathbf{0}$ in a context that is already expected to exchange some type:

$$\mathbf{0} : T \;\; \text{for any } T$$

Alternatively, we may add a subtype relation among types, give $\mathbf{0}$ a minimal type, and add a rule which allows processes with a type to appear where processes with a supertype are required [36]. We reject this approach here only because we want to explore the ideas of group-based exchange and mobility types in the simplest possible setting.

If P and Q are processes that may exchange T, then $P \mid Q$ is also such a process. Similarly for $!P$:

$$
\begin{array}{ll}
P : T, Q : T & \text{implies} \;\; P \mid Q : T \\
P : T & \text{implies} \;\; !P : T
\end{array}
$$

Therefore, by keeping track of the exchange type of a process, T-inputs and T-outputs are tracked so that they match correctly when placed in parallel.

3.2.2. Typing of Ambients

An ambient $n[P]$ is a process that exchanges nothing at the current level, so, like $\mathbf{0}$, it can be placed in parallel with any process, hence we allow it to have any exchange type:

$$n[P] : T \;\; \text{for any } T$$

There needs to be, however, a connection between the type of n and the type of P. We give to each ambient name n a type $G[T]$, meaning that n belongs to the group G and that only T exchanges are allowed in any ambient of that name. Hence, a process P can be placed inside an ambient with that name n only if the type of P is T:

$$n : G[T], P : T \;\; \text{implies} \;\; n[P] \text{ is well-formed (and can have any type)}$$

By tagging the name of an ambient with the type of exchanges, we know what kind of exchanges to expect in any ambient we enter. Moreover, we can tell what happens when we open an ambient of a given name.

3.2.3. Typing of Open

Tracking the type of I/O exchanges is not enough by itself. We also need to worry about *open*, which might open an ambient and unleash its exchanges inside the surrounding ambient.

If ambients named n permit T exchanges, then the capability $\mathit{open}\, n$ may unleash those T exchanges. We then say that $\mathit{open}\, n$ has a capability type $\mathit{Cap}[T]$, meaning that it may unleash T exchanges when used:

$$n : G[T] \quad \text{implies} \quad \mathit{open}\, n : \mathit{Cap}[T]$$

As a consequence, any process that uses a $\mathit{Cap}[T]$ must be a process that is already willing to participate in exchanges of type T, because further T exchanges may be unleashed:

$$M : \mathit{Cap}[T], P : T \quad \text{implies} \quad M.P : T$$

3.2.4. Typing of In and Out

The exercise of an *in* or *out* capability cannot cause any exchange, hence such capabilities can be prepended to any process. Following the same pattern we used with $\mathbf{0}$ and ambients, the silent nature of these capabilities is formalized by allowing them to acquire any capability type:

$$in\, n : Cap[T] \quad \text{for any } T$$
$$out\, n : Cap[T] \quad \text{for any } T$$

3.2.5. Groups

Groups are used in the exchange system to specify which kinds of messages can be exchanged inside an ambient. We add a process construct to create a new group G with scope P:

$$(\nu G)P$$

The type rule of this construct specifies that the process P should have an exchange type T that does not contain G. Then, $(\nu G)P$ can be given type T as well. That is, G is never allowed to "escape" out of the scope of (νG) into the type of $(\nu G)P$:

$$P : T, \quad G \text{ does not occur in } T \quad \text{implies} \quad (\nu G)P : T$$

4. TYPED AMBIENT CALCULUS

We are now ready for a formal presentation of the typed calculus which has been informally introduced in the previous section. We first present its syntax, then its typing rules, and finally a subject reduction theorem, which states that types are preserved during computation.

4.1. Types and Processes

Types are defined as in Section 3.1; messages and processes are defined as in Section 2, but we add the operator $(\nu G)P$ of Section 3.2.5.

Messages and Processes:

$P, Q, R ::=$	process
$(\nu G)P$	group creation
$\ldots$	as in Section 2

We identify processes up to consistent renaming of bound names and groups. In the processes $(\nu G)P$ and $(\nu n{:}W)P$, the group G and the name n, respectively, are bound, with scope P. In the process $(x_1{:}W_1, \ldots, x_k{:}W_k).P$, the names $x_1, \ldots, x_k$ are bound, with scope P.

The following table defines the free names of processes and messages, and the free groups of processes and types.

Free Names and Free Groups:

$fn((\nu G)P) \triangleq fn(P)$	$fn(n) \triangleq \{n\}$
$fn((\nu n{:}W)P) \triangleq fn(P) - \{n\}$	$fn(in\, M) \triangleq fn(M)$
$fn(\mathbf{0}) \triangleq \varnothing$	$fn(out\, M) \triangleq fn(M)$
$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$	$fn(open\, M) \triangleq fn(M)$
$fn(!P) \triangleq fn(P)$	$fn(\epsilon) \triangleq \varnothing$
$fn(M[P]) \triangleq fn(M) \cup fn(P)$	$fn(M.N) \triangleq fn(M) \cup fn(N)$

$fn(M.P) \triangleq fn(M) \cup fn(P)$
$fn((x_1{:}W_1, \ldots, x_k{:}W_k).P) \triangleq fn(P) - \{x_1, \ldots, x_k\}$
$fn(\langle M_1, \ldots, M_k \rangle) \triangleq fn(M_1) \cup \cdots \cup fn(M_k)$
$fn(go\ N.M[P]) \triangleq fn(N) \cup fn(M) \cup fn(P)$

$fg((\nu G)P) \triangleq fg(P) - \{G\}$	$fg(G[T]) \triangleq \{G\} \cup fg(T)$
$fg((\nu n{:}W)P) \triangleq fg(W) \cup fg(P)$	$fg(Cap[T]) \triangleq fg(T)$
$fg(\mathbf{0}) \triangleq \varnothing$	$fg(Shh) \triangleq \varnothing$
$fg(P \mid Q) \triangleq fg(P) \cup fg(Q)$	$fg(W_1 \times \cdots \times W_k) \triangleq$
$fg(!P) \triangleq fg(P)$	$fg(W_1) \cup \cdots \cup fg(W_k)$

$fg(M[P]) \triangleq fg(P)$
$fg(M.P) \triangleq fg(P)$
$fg((x_1{:}W_1, \ldots, x_k{:}W_k).P) \triangleq fg(W_1) \cup \cdots \cup fg(W_k) \cup fg(P)$
$fg(\langle M_1, \ldots, M_k \rangle) \triangleq \varnothing$
$fg(go\ N.M[P]) \triangleq fg(P)$

The following tables describe the operational semantics of the calculus. The type annotations present in the syntax do not play a role in reduction; they are simply carried along by the reductions.

Terms are identified up to an equivalence relation, $\equiv$, called structural congruence. This relation provides a way of rearranging processes so that interacting parts can be brought together. Then, a reduction relation, $\rightarrow$, acts on the interacting parts to produce computation steps. The core of the calculus is given by the reduction rules (Red In), (Red Out), (Red Go In), (Red Go Out), and (Red Open), for mobility, and (Red I/O), for communication.

The rules of structural congruence are the same as for the untyped ambient calculus [13], except for the addition of type annotations, and new rules for objective moves and group restriction. The rules (Struct GRes ...) describe the extrusion behavior of the (νG) binders. Note that (νG) extrudes exactly as (νn) does, hence it does not pose any dynamic restriction on the movement of ambients or messages. The rule (Struct Go ϵ) allows empty objective moves to be erased. The rules (Struct Go ϵ .), (Struct Go . ϵ), and (Struct Go . Assoc) allow the capability expression in an objective move to be re-arranged to allow application of the reduction rules (Red Go In) and (Red Go Out). (These three rules of structural congruence were missing in an earlier version of this system [10].)

Reduction:

$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.P \mid n[Q] \rightarrow P \mid Q$	(Red Open)
$\langle M_1, \ldots, M_k \rangle \mid (x_1{:}W_1, \ldots, x_k{:}W_k).P \rightarrow P\{x_1 {\leftarrow} M_1, \ldots, x_k {\leftarrow} M_k\}$	(Red I/O)
$go(in\ m.N).n[P] \mid m[Q] \rightarrow m[go\ N.n[P] \mid Q]$	(Red Go In)
$m[go(out\ m.N).n[P] \mid Q] \rightarrow go\ N.n[P] \mid m[Q]$	(Red Go Out)
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	(Red Par)

$P \to Q \Rightarrow (\nu n{:}W)P \to (\nu n{:}W)Q$	(Red Res)
$P \to Q \Rightarrow (\nu G)P \to (\nu G)Q$	(Red GRes)
$P \to Q \Rightarrow n[P] \to n[Q]$	(Red Amb)
$P' \equiv P, P \to Q, Q \equiv Q' \Rightarrow P' \to Q'$	(Red $\equiv$)

Structural Congruence:

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu n{:}W)P \equiv (\nu n{:}W)Q$	(Struct Res)
$P \equiv Q \Rightarrow (\nu G)P \equiv (\nu G)Q$	(Struct GRes)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow\, !P \equiv\, !Q$	(Struct Repl)
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Action)
$P \equiv Q \Rightarrow$ $(x_1{:}W_1, \ldots, x_k{:}W_k).P \equiv (x_1{:}W_1, \ldots, x_k{:}W_k).Q$	(Struct Input)
$P \equiv Q \Rightarrow \mathit{go}\ N.M[P] \equiv \mathit{go}\ N.M[Q]$	(Struct Go)
$P \mid Q \equiv Q \mid P$	(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)
$n_1 \neq n_2 \Rightarrow$ $(\nu n_1{:}W_1)(\nu n_2{:}W_2)P \equiv (\nu n_2{:}W_2)(\nu n_1{:}W_1)P$	(Struct Res Res)
$n \notin \mathit{fn}(P) \Rightarrow (\nu n{:}W)(P \mid Q) \equiv P \mid (\nu n{:}W)Q$	(Struct Res Par)
$n \neq m \Rightarrow (\nu n{:}W)m[P] \equiv m[(\nu n{:}W)P]$	(Struct Res Amb)
$(\nu G_1)(\nu G_2)P \equiv (\nu G_2)(\nu G_1)P$	(Struct GRes GRes)
$G \notin \mathit{fg}(W) \Rightarrow (\nu G)(\nu n{:}W)P \equiv (\nu n{:}W)(\nu G)P$	(Struct GRes Res)
$G \notin \mathit{fg}(P) \Rightarrow (\nu G)(P \mid Q) \equiv P \mid (\nu G)Q$	(Struct GRes Par)
$(\nu G)m[P] \equiv m[(\nu G)P]$	(Struct GRes Amb)
$P \mid \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n{:}W)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Res)
$(\nu G)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero GRes)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Repl)
$\epsilon.P \equiv P$	(Struct ϵ)
$(M.M').P \equiv M.M'.P$	(Struct .)
$\mathit{go}\ \epsilon.N[P] \equiv N[P]$	(Struct Go ϵ)
$\mathit{go}\ (\epsilon.M).N[P] \equiv \mathit{go}\ M.N[P]$	(Struct Go ϵ .)
$\mathit{go}\ (M.\epsilon).N[P] \equiv \mathit{go}\ M.N[P]$	(Struct Go . ϵ)
$\mathit{go}\ ((M.M').M'').N[P] \equiv \mathit{go}\ (M.(M'.M'')).N[P]$	(Struct Go . Assoc)

4.2. The Exchange Types

In the tables below, we introduce typing environments, E, the five basic judgments, and the typing rules. By convention, any antecedent of the form $E \vdash \mathcal{J}_1$, $\ldots$, $E \vdash \mathcal{J}_n$ means $E \vdash \diamond$ when $n = 0$.

Environments, E, and the Domain, $dom(E)$, of an Environment:

$E ::= \varnothing \mid E, G \mid E, n{:}W$	environment

$dom(\varnothing) \triangleq \varnothing$
$dom(E, G) \triangleq dom(E) \cup \{G\}$
$dom(E, n{:}W) \triangleq dom(E) \cup \{n\}$

Judgments:

$E \vdash \diamond$	good environment
$E \vdash W$	good message type W
$E \vdash T$	good exchange type T
$E \vdash M : W$	good message M of message type W
$E \vdash P : T$	good process P with exchange type T

Good Environments:

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

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

(Env G)
$$\frac{E \vdash \diamond \quad G \notin dom(E)}{E, G \vdash \diamond}$$

Good Types:

(Type Amb)
$$\frac{G \in dom(E) \quad E \vdash T}{E \vdash G[T]}$$

(Type Cap)
$$\frac{E \vdash T}{E \vdash Cap[T]}$$

(Type Shh)
$$\frac{E \vdash \diamond}{E \vdash Shh}$$

(Type Prod)
$$\frac{E \vdash W_1 \quad \cdots \quad E \vdash W_k}{E \vdash W_1 \times \cdots \times W_k}$$

Good Messages:

(Exp n)
$$\frac{E', n{:}W, E'' \vdash \diamond}{E', n{:}W, E'' \vdash n : W}$$

(Exp .)
$$\frac{E \vdash M : Cap[T] \quad E \vdash M' : Cap[T]}{E \vdash M.M' : Cap[T]}$$

(Exp ϵ)
$$\frac{E \vdash Cap[T]}{E \vdash \epsilon : Cap[T]}$$

(Exp In)
$$\frac{E \vdash n : G[S] \quad E \vdash T}{E \vdash in\, n : Cap[T]}$$

(Exp Out)
$$\frac{E \vdash n : G[S] \quad E \vdash T}{E \vdash out\, n : Cap[T]}$$

(Exp Open)
$$\frac{E \vdash n : G[T]}{E \vdash open\, n : Cap[T]}$$

Good Processes:

(Proc Action)
$$\frac{E \vdash M : Cap[T] \quad E \vdash P : T}{E \vdash M.P : T}$$

(Proc Amb)
$$\frac{E \vdash M : G[S] \quad E \vdash P : S \quad E \vdash T}{E \vdash M[P] : T}$$

(Proc Res)
$$\frac{E, n{:}G[S] \vdash P : T}{E \vdash (\nu n{:}G[S])P : T}$$

(Proc GRes)
$$\frac{E, G \vdash P : T \quad G \notin fg(T)}{E \vdash (\nu G)P : T}$$

(Proc Zero)
$$\frac{E \vdash T}{E \vdash \mathbf{0} : T}$$

(Proc Par)
$$\frac{E \vdash P : T \quad E \vdash Q : T}{E \vdash P \mid Q : T}$$

(Proc Repl)
$$\frac{E \vdash P : T}{E \vdash !P : T}$$

(Proc Input)
$$\frac{E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P : W_1 \times \cdots \times W_k}{E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).P : W_1 \times \cdots \times W_k}$$

(Proc Output)
$$\frac{E \vdash M_1 : W_1 \quad \cdots \quad E \vdash M_k : W_k}{E \vdash \langle M_1, \ldots, M_k \rangle : W_1 \times \cdots \times W_k}$$

(Proc Go)
$$\frac{E \vdash N : Cap[Shh] \quad E \vdash M : G[S] \quad E \vdash P : S \quad E \vdash T}{E \vdash go\ N.M[P] : T}$$

4.3. Subject Reduction

We obtain a standard subject reduction result. A subtle point, though, is the need to account for the appearance of new groups ($G_1, \ldots, G_k$, below) during reduction. This is because reduction is defined up to structural congruence, and structural congruence does not preserve the set of free groups of a process. The culprit is the rule $(\nu n{:}W)\mathbf{0} \equiv \mathbf{0}$, in which groups free in W are not free in $\mathbf{0}$.

THEOREM 4.1 (Subject Congruence). *If $E \vdash P : T$ and $P \equiv Q$ then there are $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : T$.*

Proof. See Appendix A. ■

THEOREM 4.2 (Subject Reduction). *If $E \vdash P : T$ and $P \to Q$ then there are $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : T$.*

Proof. See Appendix A. ■

Subject reduction specifies that, if P is well-typed, it will only reduce to well-typed terms. This fact has some practical consequences:

- P will never reduce to meaningless processes allowed by the syntax like $(in\, n)[P]$;
- no process deriving from P will contain an ambient where a process attempts an input or output operation which does not match the ambient type.

Subject reduction has also interesting and subtle connections with secrecy of names.

Consider a well-typed process $((\nu G)P) \mid O$, where O is a type-checked "opponent", and a name n is declared inside P with a type $G[T]$. Although (νG) can be extruded arbitrarily far, according to the extrusion rules, no process which derives from the opponent O will ever be able to read n through an input $(x{:}W).Q$. Any process $\langle n \rangle \mid (x{:}W).Q$ which derives from $((\nu G)P) \mid O$ is well-typed, hence $W = G[T]$, but the opponent was not, by assumption, in the initial scope of G, and therefore cannot even mention the type $G[T]$. Therefore, we can guarantee that names of group G can never be communicated to processes outside the initial scope of G, simply because those processes cannot name G to receive the message. (Elsewhere [11] we extend this argument to the case of untyped opponents.)

This situation is in sharp contrast with ordinary name restriction, where a name that is initially held secret (e.g. a key) may accidentally be given away and misused (e.g. to decrypt current or old messages). This is because scoping of names can be extruded too far, inadvertently. Scoping of groups can be extruded as well, but still offers protection against accidental or even malicious leakage.

Of course, we would have even stronger protection if we did not allow (νG) binders to extrude at all. But this would be too rigid. Since (νG) binders can be extruded, they do not impede the mobility of ambients that carry secrets. They only prevent those ambients from giving the secrets away. Consider the following example of traveling agents sharing secrets.

$$\begin{array}{l} a[(\nu G)(\nu k' : G[\mathit{Shh}])(\nu k'' : G[\mathit{Shh}])(\\ \qquad\qquad\qquad k'[\mathit{out}\; a.\mathit{in}\; b.\mathit{out}\; b.\mathit{in}\; c] \mid \\ \qquad\qquad\qquad k''[\mathit{out}\; a.\mathit{in}\; c.\mathit{in}\; k']) \\ \quad] \mid b[] \mid c[] \end{array}$$

Within an ambient a, two agents share a secret group G and two names k' and k'' belonging to that group. The two agents adopt the names k' and k'' as their respective names, knowing that those names cannot be leaked even by themselves. This way, as they travel, nobody else can interfere with them. If somebody interferes with them, or demonstrates knowledge of the names k' or k'', the agents know that the other party must be (a descendant of) the other agent. In this example, the first agent travels to ambient b and then to c, and the second agent goes to ambient c directly. The scope extrusion rules for groups and names allow this to happen. Inside c, out of the intial scope of (νG), the second agent then interacts with the first by entering it. It can do so because it still holds the shared secret k'.

The proof that group extrusion preserves types can be found in the appendix, but we comment here on the crucial case: the preservation of typing by the extrusion rule (Struct GRes Amb).

For a well-typed P, $(\nu G)P$ is well-typed if and only if P does not communicate a tuple which names G in its type (rule (Proc GRes)): (νG) must not "see" G-typed names communicated at its own level. This intuition suggests that, referring to the following table, P' should be typeable ((νG) cannot "see" the output $\langle n\rangle$) while P'' should be not ($\langle n\rangle$ is at the same level as (νG)). However, the two processes are equivalent, modulo extrusion of (νG) (rule (Struct GRes Amb)):

$$\begin{array}{rcl} P' & = & (\nu G)m[(\nu n{:}G[\mathit{Shh}])\langle n\rangle] \\ P'' & = & m[(\nu G)(\nu n{:}G[\mathit{Shh}])\langle n\rangle] \end{array}$$

We go through the example step by step, to solve the apparent paradox. First consider the term

$$(\nu G)(\nu n{:}G[\mathit{Shh}])\langle n\rangle$$

This term cannot be typed, because G attempts to escape the scope of (νG) as the type of the message n. An attempted typing derivation fails at the last step below

$$\begin{array}{ll} \ldots & \\ \Rightarrow & G, n{:}G[\mathit{Shh}] \vdash n : G[\mathit{Shh}] \\ \Rightarrow & G, n{:}G[\mathit{Shh}] \vdash \langle n\rangle : G[\mathit{Shh}] \\ \Rightarrow & G \vdash (\nu n{:}G[\mathit{Shh}])\langle n\rangle : G[\mathit{Shh}] \\ \not\Rightarrow & \vdash (\nu G)(\nu n{:}G[\mathit{Shh}])\langle n\rangle : G[\mathit{Shh}] \quad (\text{because } G \in \mathit{fg}(G[\mathit{Shh}])) \end{array}$$

Similarly, the term

$$(\nu m{:}W)m[(\nu G)(\nu n{:}G[\mathit{Shh}])\langle n\rangle]$$

cannot be typed, because it contains the previous untypeable term. But now consider the following term, which is equivalent to the one above up to structural congruence, by extrusion of (νG) across an ambient boundary:

$$(\nu m{:}W)(\nu G)m[(\nu n{:}G[\mathit{Shh}])\langle n\rangle]$$

This term might appear typeable (contradicting the subject congruence property) because the message $\langle n\rangle{:}G[\mathit{Shh}]$ is confined to the ambient m, and $m[\ldots]$ can be given an arbitrary type, e.g. Shh, which does not contain G. Therefore (νG) would not "see" any occurrence of G escaping from its scope. However, consider the type of m in this term. It must have the form $H[T]$, where H is some group, and T is the type of messages exchanged inside m. But that's $G[\mathit{Shh}]$. So we would have

$$(\nu m{:}H[G[\mathit{Shh}]])(\nu G)m[(\nu n{:}G[\mathit{Shh}])\langle n\rangle]$$

which is not typeable because the first occurrence of G is out of scope.

This example tells us why (νG) *intrusion* (floating inwards) into ambients is not going to break good typing: (νG) cannot enter the scope of the $(\nu m{:}W)$ restriction which creates the name m of an ambient where messages with a G-named type are exchanged. This prevents (νG) from entering such ambients.

Indeed, the following variation (not equivalent to the previous one) is typeable, but (νG) cannot intrude any more:

$$(\nu G)(\nu m{:}H[G[\mathit{Shh}]])m[(\nu n{:}G[\mathit{Shh}])\langle n\rangle]$$

5. OPENING CONTROL

Ambient opening is a prerequisite for any communication to happen between processes which did not originate in the same ambient, as exemplified by any channel encoding.

On the other hand, opening is one of the most delicate operations in the ambient calculus, since the contents of the guest spill inside the host, with two different classes of possible consequences:

- the content of the guest acquires the possibility of performing communications inside the hosts, and of moving the host around;
- the host is now able to examine the content of the guest, mainly in terms of receiving messages sent by the processes inside the guest, and of opening its sub-ambients.

For these reasons, a type system for ambients should support a careful control of the usage of the *open* capability.

5.1. The System

In this section, we enrich the ambient types, $G[T]$, and the capability types, $Cap[T]$, of the previous type system to control usage of the *open* capability.

To control the opening of ambients, we formalize the constraint that the name of any ambient opened by a process is in one of the groups $G_1, \ldots, G_k$, but in no others. To do so, we add an attribute $°\{G_1, \ldots, G_k\}$ to ambient types, which now take the form $G[°\{G_1, \ldots, G_k\}, T]$. A name of this type is in group G, and names ambients within which processes may exchange messages of type T and may only open ambients in the groups $G_1, \ldots, G_k$. We need to add the same attribute to capability types, which now take the form $Cap[°\{G_1, \ldots, G_k\}, T]$. Exercising a capability of this type may unleash exchanges of type T and openings of ambients in groups $G_1, \ldots, G_k$. The typing judgment for processes acquires the form $E \vdash P : °\{G_1, \ldots, G_k\}, T$. The pair $°\{G_1, \ldots, G_k\}, T$ constrains both the *opening effects* (what ambients the process opens) and the *exchange effects* (what messages the process exchanges). We call such a pair an *effect*, and introduce the metavariable F to range over effects. It is also convenient to introduce metavariables $\mathbf{G}$, $\mathbf{H}$ to range over finite sets of groups. The following tables summarize these metavariable conventions and our enhanced syntax for types:

Group Sets:

$\mathbf{G}, \mathbf{H} ::= \{G_1, \ldots, G_k\}$	finite set of groups

Types:

$W ::=$	message type
$G[F]$	name in group G for ambients which contain processes with F effects
$Cap[F]$	capability (unleashes F effects)
$F ::=$	effect
$°\mathbf{H}, T$	may open $\mathbf{H}$, may exchange T
$S, T ::=$	exchange type

Shh	no exchange
$W_1 \times \cdots \times W_k$	tuple exchange

The definition of free groups is the same as in Section 4 except that we redefine $fg(W)$ by the equations $fg(G[F]) = \{G\} \cup fg(F)$ and $fg(Cap[F]) = fg(F)$, and we define $fg(F) = \mathbf{H} \cup fg(T)$ where $F = {}^{\circ}\mathbf{H}, T$.

The following tables define the type system in detail. There are five basic judgments as before. They have the same format except that the judgment $E \vdash F$, meaning that the effect F is good given environment E, replaces the previous judgment $E \vdash T$. We omit the three rules for deriving good environments; they are exactly as in the previous section. There are two main differences between the other rules below and the rules of the previous section. First, effects, F, replace exchange types, T, throughout. Second, in the rule (Exp Open), the condition $G \in \mathbf{H}$ constrains the opening effect $\mathbf{H}$ of a capability *open* n to include the group G, the group of the name n.

Judgments:

$E \vdash \diamond$	good environment
$E \vdash W$	good message type W
$E \vdash F$	good effect F
$E \vdash M : W$	good message M of message type W
$E \vdash P : F$	good process P with F effects

Good Types:

(Type Amb)
$$\frac{G \in dom(E) \quad E \vdash F}{E \vdash G[F]}$$

(Type Cap)
$$\frac{E \vdash F}{E \vdash Cap[F]}$$

(Effect Shh)
$$\frac{\mathbf{H} \subseteq dom(E) \quad E \vdash \diamond}{E \vdash {}^{\circ}\mathbf{H}, Shh}$$

(Effect Prod)
$$\frac{\mathbf{H} \subseteq dom(E) \quad E \vdash W_1 \quad \cdots \quad E \vdash W_k}{E \vdash {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k}$$

Good Messages:

(Exp n)
$$\frac{E', n{:}W, E'' \vdash \diamond}{E', n{:}W, E'' \vdash n : W}$$

(Exp ϵ)
$$\frac{E \vdash Cap[F]}{E \vdash \epsilon : Cap[F]}$$

(Exp .)
$$\frac{E \vdash M : Cap[F] \quad E \vdash M' : Cap[F]}{E \vdash M.M' : Cap[F]}$$

(Exp In)
$$\frac{E \vdash n : G[F] \quad E \vdash {}^{\circ}\mathbf{H}, T}{E \vdash in\, n : Cap[{}^{\circ}\mathbf{H}, T]}$$

(Exp Out)
$$\frac{E \vdash n : G[F] \quad E \vdash {}^{\circ}\mathbf{H}, T}{E \vdash \mathit{out}\, n : \mathit{Cap}[{}^{\circ}\mathbf{H}, T]}$$

(Exp Open)
$$\frac{E \vdash n : G[{}^{\circ}\mathbf{H}, T] \quad G \in \mathbf{H}}{E \vdash \mathit{open}\, n : \mathit{Cap}[{}^{\circ}\mathbf{H}, T]}$$

Good Processes:

(Proc Action)
$$\frac{E \vdash M : \mathit{Cap}[F] \quad E \vdash P : F}{E \vdash M.P : F}$$

(Proc Amb)
$$\frac{E \vdash M : G[F] \quad E \vdash P : F \quad E \vdash F'}{E \vdash M[P] : F'}$$

(Proc Res)
$$\frac{E, n{:}G[F] \vdash P : F'}{E \vdash (\nu n{:}G[F])P : F'}$$

(Proc GRes)
$$\frac{E, G \vdash P : F \quad G \notin \mathit{fg}(F)}{E \vdash (\nu G)P : F}$$

(Proc Zero)
$$\frac{E \vdash F}{E \vdash \mathbf{0} : F}$$

(Proc Par)
$$\frac{E \vdash P : F \quad E \vdash Q : F}{E \vdash P \mid Q : F}$$

(Proc Repl)
$$\frac{E \vdash P : F}{E \vdash !P : F}$$

(Proc Input)
$$\frac{E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P : {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k}{E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).P : {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k}$$

(Proc Output)
$$\frac{E \vdash M_1 : W_1 \quad \cdots \quad E \vdash M_k : W_k \quad \mathbf{H} \subseteq \mathit{dom}(E)}{E \vdash \langle M_1, \ldots, M_k \rangle : {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k}$$

(Proc Go)
$$\frac{E \vdash N : \mathit{Cap}[{}^{\circ}\{\}, \mathit{Shh}] \quad E \vdash M : G[F] \quad E \vdash P : F \quad E \vdash F'}{E \vdash \mathit{go}\; N.M[P] : F'}$$

5.2. Subject Reduction

We obtain a subject reduction result.

THEOREM 5.1. *If $E \vdash P : F$ and $P \to Q$ then there are $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : F$.*

Proof. See the appendix. ■

Here is a simple example of a typing derivable in this system:

$$G, n{:}G[{}^{\circ}\{G\}, \mathit{Shh}] \vdash n[\mathbf{0}] \mid \mathit{open}\; n.\mathbf{0} : {}^{\circ}\{G\}, \mathit{Shh}$$

This asserts that the whole process $n[\mathbf{0}] \mid \mathit{open}\; n.\mathbf{0}$ is well-typed and opens only ambients in the group G.

On the other hand, one might expect the following variant to be derivable, but it is not:

$$G, n{:}G[{}^{\circ}\{\}, \mathit{Shh}] \vdash n[\mathbf{0}] \mid \mathit{open}\, n.\mathbf{0} : {}^{\circ}\{G\}, \mathit{Shh}$$

This is because the typing rule (Exp Open) requires the effect unleashed by the *open* n capability to be the same as the effect contained within the ambient n. But the opening effect ${}^{\circ}\{\}$ specified by the type $G[{}^{\circ}\{\}, \mathit{Shh}]$ of n cannot be the same as the effect unleashed by *open* n, because (Exp Open) also requires the latter to at least include the group G of n.

This feature of (Exp Open) has a positive side-effect: the type $G[{}^{\circ}\mathbf{G}, T]$ of an ambient name n not only tells which opening effects may happen inside the ambient, but also tells whether n may be opened from outside: it is openable only if $G \in \mathbf{G}$, since this is the only case when $\mathit{open}\, n.\mathbf{0} \mid n[P]$ may be well-typed. Hence, the presence of G in the set $\mathbf{G}$ may either mean that n is meant to be an ambient within which other ambients in group G may be opened, or that it is meant to be an openable ambient.

More generally, because of the shape of the open rule, the opening effects in the ambient type of n not only record the openings that may take place inside the ambient, but also the opening effects of any ambient m which is going to open n, and, recursively, of any ambient which is going to open m as well. A similar phenomenon occurs with exchange types and with the subjective-crossing effects of the next section.

While this turns out to be unproblematic for the examples we consider in this paper, one may prefer to avoid this "inward propagation" of effects by replacing (Exp Open) with the following rule:

$$\frac{E \vdash n : G[{}^{\circ}\mathbf{H}, T]}{E \vdash \mathit{open}\, n : \mathit{Cap}[{}^{\circ}(\{G\} \cup \mathbf{H}), T]}$$

With this rule, we could derive that the example process above, $n[\mathbf{0}] \mid \mathit{open}\, n.\mathbf{0}$, has effect ${}^{\circ}\{G\}, \mathit{Shh}$, with no need of attributing this effect to processes running inside n itself, but unfortunately, subject reduction fails. To see this, consider the process $\mathit{open}\, n \mid n[\mathit{open}\, m]$, which can be assigned the effect ${}^{\circ}\{G, H\}, \mathit{Shh}$:

$$G, H, m{:}G[{}^{\circ}\{\}, \mathit{Shh}], n{:}H[{}^{\circ}\{G\}, \mathit{Shh}] \vdash \mathit{open}\, n \,|\, n[\mathit{open}\, m] : {}^{\circ}\{G, H\}, \mathit{Shh}$$

The process reduces in one step to *open* m, but we cannot derive the following:

$$G, H, m{:}G[{}^{\circ}\{\}, \mathit{Shh}], n{:}H[{}^{\circ}\{G\}, \mathit{Shh}] \vdash \mathit{open}\, m : {}^{\circ}\{G, H\}, \mathit{Shh}$$

To obtain a subject reduction property in the presence of the rule displayed above, we should introduce a notion of subtyping, such that if $\mathbf{G} \subseteq \mathbf{H}$ and a process has type ${}^{\circ}\mathbf{G}, T$, then the process has type ${}^{\circ}\mathbf{H}, T$ too. This would complicate the type system, as shown in [36]. Moreover, we would lose the indirect way of declaring ambient openability, so we prefer to stick to the basic approach.

6. CROSSING CONTROL

This section presents the third and final type system of the paper. We obtain it by enriching the type system of Section 5 with attributes to control the mobility of ambients.

6.1. The System

Movement operators enable an ambient n to cross the boundary of another ambient m either by entering it via an $\mathit{in}\, m$ capability or by exiting it via an $\mathit{out}\, m$ capability. In the type system of this section, the type of n lists those groups that may be crossed; the ambient n may only cross the boundary of another ambient m if the group of m is included in this list. In our typed calculus, there are two kinds of movement, subjective moves and objective moves, for reasons explained in Section 6.2. Therefore, we separately list those groups that may be crossed by objective moves and those groups that may be crossed by subjective moves.

We add new attributes to the syntax of ambient types, effects, and capability types. An ambient type acquires the form $G \frown \mathbf{G}'[\frown\mathbf{G}, \circ\mathbf{H}, T]$. An ambient of this type is in group G, may cross ambients in groups $\mathbf{G}'$ by objective moves, may cross ambients in groups $\mathbf{G}$ by subjective moves, may open ambients in groups $\mathbf{H}$, and may contain exchanges of type T. An effect, F, of a process is now of the form $\frown\mathbf{G}, \circ\mathbf{H}, T$. It asserts that the process may exercise *in* and *out* capabilities to accomplish subjective moves across ambients in groups $\mathbf{G}$, that the process may open ambients in groups $\mathbf{H}$, and that the process may exchange messages of type T. Finally, a capability type retains the form $\mathit{Cap}[F]$, but with the new interpretation of F. Exercising a capability of this type may unleash F effects.

Types:

$W ::=$	message type
$G \frown \mathbf{G}[F]$	name in group G for ambients which cross $\mathbf{G}$ objectively and contain processes with F effects
$\mathit{Cap}[F]$	capability (unleashes F effects)
$F ::=$	effect
$\frown\mathbf{G}, \circ\mathbf{H}, T$	crosses $\mathbf{G}$, opens $\mathbf{H}$, exchanges T
$S, T ::=$	exchange type
Shh	no exchange
$W_1 \times \cdots \times W_k$	tuple exchange

The definition of free groups is the same as in Section 4 except that we redefine $\mathit{fg}(W)$ by the equations $\mathit{fg}(G \frown \mathbf{G}[F]) = \{G\} \cup \mathbf{G} \cup \mathit{fg}(F)$ and $\mathit{fg}(\mathit{Cap}[F]) = \mathit{fg}(F)$, and we define $\mathit{fg}(F) = \mathbf{G} \cup \mathbf{H} \cup \mathit{fg}(T)$ where $F = \frown\mathbf{G}, \circ\mathbf{H}, T$.

The format of the five judgments making up the system is the same as in Section 5. We omit the three rules defining good environments; they are as in Section 4. There are two main changes to the previous system to control mobility. First, (Exp In) and (Exp Out) change to assign a type $\mathit{Cap}[\frown\mathbf{G}, \circ\mathbf{H}, T]$ to capabilities $\mathit{in}\, n$ and $\mathit{out}\, n$ only if $G \in \mathbf{G}$ where G is the group of n. Second, (Proc Go) changes to allow an objective move of an ambient of type $G \frown \mathbf{G}'[F]$ by a capability of type $\mathit{Cap}[\frown\mathbf{G}, \circ\mathbf{H}, T]$ only if $\mathbf{G} = \mathbf{G}'$.

Good Types:

(Type Amb)
$$\frac{G \in dom(E) \quad \mathbf{G} \subseteq dom(E) \quad E \vdash F}{E \vdash G \frown \mathbf{G}[F]}$$

(Type Cap)
$$\frac{E \vdash F}{E \vdash \mathit{Cap}[F]}$$

(Effect Shh)
$$\frac{\mathbf{G} \subseteq dom(E) \quad \mathbf{H} \subseteq dom(E) \quad E \vdash \diamond}{E \vdash \frown\mathbf{G}, {}^{\circ}\mathbf{H}, \mathit{Shh}}$$

(Effect Prod)
$$\frac{\mathbf{G} \subseteq dom(E) \quad \mathbf{H} \subseteq dom(E) \quad E \vdash W_1 \quad \cdots \quad E \vdash W_k}{E \vdash \frown\mathbf{G}, {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k}$$

Good Messages:

(Exp n)
$$\frac{E', n{:}W, E'' \vdash \diamond}{E', n{:}W, E'' \vdash n : W}$$

(Exp ϵ)
$$\frac{E \vdash \mathit{Cap}[F]}{E \vdash \epsilon : \mathit{Cap}[F]}$$

(Exp .)
$$\frac{E \vdash M : \mathit{Cap}[F] \quad E \vdash M' : \mathit{Cap}[F]}{E \vdash M.M' : \mathit{Cap}[F]}$$

(Exp In)
$$\frac{E \vdash n : G \frown \mathbf{G}'[F] \quad E \vdash \frown\mathbf{G}, {}^{\circ}\mathbf{H}, T \quad G \in \mathbf{G}}{E \vdash \mathit{in}\, n : \mathit{Cap}[\frown\mathbf{G}, {}^{\circ}\mathbf{H}, T]}$$

(Exp Out)
$$\frac{E \vdash n : G \frown \mathbf{G}'[F] \quad E \vdash \frown\mathbf{G}, {}^{\circ}\mathbf{H}, T \quad G \in \mathbf{G}}{E \vdash \mathit{out}\, n : \mathit{Cap}[\frown\mathbf{G}, {}^{\circ}\mathbf{H}, T]}$$

(Exp Open)
$$\frac{E \vdash n : G \frown \mathbf{G}'[\frown\mathbf{G}, {}^{\circ}\mathbf{H}, T] \quad G \in \mathbf{H}}{E \vdash \mathit{open}\, n : \mathit{Cap}[\frown\mathbf{G}, {}^{\circ}\mathbf{H}, T]}$$

Good Processes:

(Proc Action)
$$\frac{E \vdash M : \mathit{Cap}[F] \quad E \vdash P : F}{E \vdash M.P : F}$$

(Proc Amb)
$$\frac{E \vdash M : G \frown \mathbf{G}[F] \quad E \vdash P : F \quad E \vdash F'}{E \vdash M[P] : F'}$$

(Proc Res)
$$\frac{E, n{:}G \frown \mathbf{G}[F] \vdash P : F'}{E \vdash (\nu n{:}G \frown \mathbf{G}[F])P : F'}$$

(Proc GRes)
$$\frac{E, G \vdash P : F \quad G \notin \mathit{fg}(F)}{E \vdash (\nu G)P : F}$$

(Proc Zero)
$$\frac{E \vdash F}{E \vdash \mathbf{0} : F}$$

(Proc Par)
$$\frac{E \vdash P : F \quad E \vdash Q : F}{E \vdash P \mid Q : F}$$

(Proc Repl)
$$\frac{E \vdash P : F}{E \vdash !P : F}$$

(Proc Input)
$$\frac{E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P : {}^\frown\mathbf{G}, {}^\circ\mathbf{H}, W_1 \times \cdots \times W_k}{E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).P : {}^\frown\mathbf{G}, {}^\circ\mathbf{H}, W_1 \times \cdots \times W_k}$$

(Proc Output)
$$\frac{E \vdash M_1 : W_1 \quad \cdots \quad E \vdash M_k : W_k \quad \mathbf{G} \subseteq dom(E) \quad \mathbf{H} \subseteq dom(E)}{E \vdash \langle M_1, \ldots, M_k \rangle : {}^\frown\mathbf{G}, {}^\circ\mathbf{H}, W_1 \times \cdots \times W_k}$$

(Proc Go)
$$\frac{E \vdash N : \mathit{Cap}[{}^\frown\mathbf{G}, {}^\circ\{\}, \mathit{Shh}] \quad E \vdash M : G\,{}^\frown\mathbf{G}[F] \quad E \vdash P : F \quad E \vdash F'}{E \vdash \mathit{go}\; N.M[P] : F'}$$

THEOREM 6.1. *If* $E \vdash P : F$ *and* $P \to Q$ *then there are* $G_1, \ldots, G_k$ *such that* $G_1, \ldots, G_k, E \vdash Q : F$.

Proof. See the appendix. ■

6.2. The Need for Objective Moves

We can now show how primitive typing rules for objective moves allow us to assign better types in some crucial situations. Recall the untyped example from Section 2. Suppose we have two groups *Ch* and *Pk* (for channels and packets). Let W be any well-formed type (where *Ch* and *Pk* may appear), and set P to be the example process:

$$P \;=\; a[p[\mathit{out}\; a.\mathit{in}\; b.\langle c\rangle]] \mid b[\mathit{open}\; p.(x{:}W).x[]]$$

Let

$$\begin{aligned} E \;=\; & \mathit{Ch}, \mathit{Pk}, \\ & a{:}\mathit{Ch}\,{}^\frown\{\}[{}^\frown\{\}, {}^\circ\{\}, \mathit{Shh}], \\ & b{:}\mathit{Ch}\,{}^\frown\{\}[{}^\frown\{\mathit{Ch}\}, {}^\circ\{\mathit{Pk}\}, W], \\ & c{:}W, \\ & p{:}\mathit{Pk}\,{}^\frown\{\}[{}^\frown\{\mathit{Ch}\}, {}^\circ\{\mathit{Pk}\}, W] \end{aligned}$$

and we can derive the typings:

$$\begin{aligned} & E \vdash \mathit{out}\; a.\mathit{in}\; b.\langle c\rangle : {}^\frown\{\mathit{Ch}\}, {}^\circ\{\mathit{Pk}\}, W \\ & E \vdash \mathit{open}\; p.(x{:}W).x[] : {}^\frown\{\mathit{Ch}\}, {}^\circ\{\mathit{Pk}\}, W \\ & E \vdash P : {}^\frown\{\}, {}^\circ\{\}, \mathit{Shh} \end{aligned}$$

From the typing $a : \mathit{Ch}\,{}^\frown\{\}[{}^\frown\{\}, {}^\circ\{\}, \mathit{Shh}]$, we can tell that a is an immobile ambient in which nothing is exchanged and that cannot be opened. From the typings $p{:}\mathit{Pk}\,{}^\frown\{\}[{}^\frown\{\mathit{Ch}\}, {}^\circ\{\mathit{Pk}\}, W]$, $b{:}\mathit{Ch}\,{}^\frown\{\}[{}^\frown\{\mathit{Ch}\}, {}^\circ\{\mathit{Pk}\}, W]$, we can tell that the ambients b and p cross only Ch ambients, open only Pk ambients, and contain W exchanges; the typing of p also tells us it can be opened. This is not fully satisfactory, since, if b were meant to be immobile, we would like to express this immobility invariant in its type. However, since b opens a subjectively mobile ambient, then b must be typed as if it were subjectively mobile itself. The problem is quite general, as it applies to any immobile ambient wishing to open a subjectively mobile one.

This problem can be solved by replacing the subjective moves by objective moves, since objective moves are less expressive than subjective moves, but they cannot be inherited by opening another ambient. Let Q be the example process with objective instead of subjective moves:

$$Q \;=\; a[go(out\ a.in\ b).p[\langle c\rangle]] \mid b[open\ p.(x{:}W).x[]]$$

Let

$$\begin{aligned} E \;=\;& Ch, Pk,\\ & a{:}Ch^{\curvearrowright}\{\}[^{\curvearrowright}\{\},^{\circ}\{\}, Shh],\\ & b{:}Ch^{\curvearrowright}\{\}[^{\curvearrowright}\{\},^{\circ}\{Pk\}, W],\\ & c{:}W,\\ & p{:}Pk^{\curvearrowright}\{Ch\}[^{\curvearrowright}\{\},^{\circ}\{Pk\}, W] \end{aligned}$$

and we can derive:

$$\begin{aligned} & E \vdash out\ a.in\ b : Cap[^{\curvearrowright}\{Ch\},^{\circ}\{\}, Shh]\\ & E \vdash go(out\ a.in\ b).p[\langle c\rangle] : {}^{\curvearrowright}\{\},^{\circ}\{\}, Shh\\ & E \vdash open\ p.(x{:}W).x[] : {}^{\curvearrowright}\{\},^{\circ}\{Pk\}, W\\ & E \vdash Q : {}^{\curvearrowright}\{\},^{\circ}\{\}, Shh \end{aligned}$$

The typings of a and c are unchanged, but the new typings of p and b are more informative. We can tell from the typing $p{:}Pk^{\curvearrowright}\{Ch\}[^{\curvearrowright}\{\},^{\circ}\{Pk\}, W]$ that movement of p is due to objective rather than subjective moves. Moreover, as desired, we can tell from the typing $b{:}Ch^{\curvearrowright}\{\}[^{\curvearrowright}\{\},^{\circ}\{Pk\}, W]$ that the ambient b is immobile.

This example suggests that in some situations objective moves lead to more informative typings than subjective moves. Still, subjective moves are essential for moving ambients containing running processes. An extended example in Section 8 illustrates the type system of this section; the treatment of thread mobility makes essential use of subjective moves.

6.3. Relationship to Binary Annotations

The system of this section generalizes our previous system of binary locking and mobility annotations [9]. In that system, the type of a name takes the form $Amb^{Y\,Z_o}[^{Z_s}T]$, where the *locking annotation*, Y, is either *locked*, $\bullet$, or *unlocked*, $\circ$, and the *mobility annotations*, Z_o and Z_s, are each either *mobile*, $\curvearrowright$, or *immobile*, $\veebar$. An ambient of this type may be opened if and only if $Y = \circ$, it may be moved objectively if and only if $Z_o = \curvearrowright$, and it may be moved subjectively if and only if $Z_s = \curvearrowright$.

That system can be understood as a degenerate form of the current one, where we only use two groups, L (for *Locked*) and U (for *Unlocked*), so that any ambient name will belong to one of these two groups. Then we understand a type $Amb^{Y\ Z_o}[{}^{Z_s}T]$ as a type $G \frown \mathbf{G}_o[\frown \mathbf{G}_s, {}^{\circ}\mathbf{H}, T']$ as follows:

- If the objective mobility annotation Z_o is $\curvearrowright$ (mobile), let $\mathbf{G}_o = \frown\{L, U\}$ (may cross any ambient). If the objective mobility annotation Z_o is $\veebar$ (immobile), let $\mathbf{G}_o = \frown\{\}$ (may cross nothing).
- We translate the subjective mobility annotation Z_s to the effect $\mathbf{G}_s$ in the same way.
- If the locking annotation Y is $\bullet$ (locked), let $G = L$ and $\mathbf{H} = \{U\}$ (locked, may open any unlocked ambient). If the locking annotation Y is $\circ$ (unlocked), let $G = U$ and $\mathbf{H} = \{U\}$ (unlocked, may be opened and may open any unlocked ambient).

It is then straightforward to show that $\vdash P : T$ holds in the system of [9] iff $L,\ U \vdash \text{translate}(P) : \text{translate}(T)$ holds in the system of this section, where *translate* translates T and the types in P as specified above.

7. EFFECT SAFETY

Like most other type systems for concurrent calculi, ours does not guarantee liveness properties, for example, the absence of deadlocks. Still, we may regard the effect assigned to a process as a safety property: an upper bound on the capabilities that may be exercised by the process, and hence on its behavior. We formalize this idea in the setting of our third type system, and explain some consequences.

We say that a process P exercises a capability M, one of *in* n or *out* n or *open* n, just if $P \downarrow M$ may be derived by the following rules:

Exercising a Capability: $P \downarrow M$ **where** $M \in \{in\ n, out\ n, open\ n\}$

(Ex Cap)	(Ex Par 1)	(Ex Par 2)	(Ex Res)	(Ex ResG)
$P \equiv M.Q$	$P \downarrow M$	$Q \downarrow M$	$P \downarrow M \quad n \notin fn(M)$	$P \downarrow M$
$P \downarrow M$	$P \mid Q \downarrow M$	$P \mid Q \downarrow M$	$(\nu n{:}W)P \downarrow M$	$(\nu G)P \downarrow M$

The following asserts that the group of the name contained in any capability exercised by a well-typed process is bounded by the effect assigned to the process. We give the proof in Appendix B.

PROPOSITION 7.1 (Effect Safety). *Suppose that* $E \vdash P : \frown\mathbf{G}, {}^{\circ}\mathbf{H}, T$.

(1)*If* $P \downarrow in\ n$ *then* $E \vdash n : G \frown \mathbf{G}'[F]$ *for some type* $G \frown \mathbf{G}'[F]$ *with* $G \in \mathbf{G}$.
(2)*If* $P \downarrow out\ n$ *then* $E \vdash n : G \frown \mathbf{G}'[F]$ *for some type* $G \frown \mathbf{G}'[F]$ *with* $G \in \mathbf{G}$.
(3)*If* $P \downarrow open\ n$ *then* $E \vdash n : G \frown \mathbf{G}'[F]$ *for some type* $G \frown \mathbf{G}'[F]$ *with* $G \in \mathbf{H}$.

To explain the intuitive significance of this proposition, consider a name $m : H \frown \mathbf{H}'[\frown\mathbf{G}, {}^{\circ}\mathbf{H}, T]$ and a well-typed ambient $m[P]$. Suppose that $m[P]$ is a sub-process of some well-typed process Q. We can make two connections between the capabilities exhibited by the process P and the reductions immediately derivable from the whole process Q. First, within Q, the ambient $m[P]$ can immediately cross (via subjective moves) the boundary of another ambient named n of some group

G only if either $P \downarrow in\, n$ or $P \downarrow out\, n$. The typing rule for ambients implies that P must have effect $^{\frown}\mathbf{G},^{\circ}\mathbf{H}, T$. Part (1) or (2) of the proposition implies that the set $\mathbf{G}$ contains G. Second, suppose that P includes a top-level ambient named n. The boundary of n can be immediately dissolved only if $P \downarrow open\, n$. Since P has effect $^{\frown}\mathbf{G},^{\circ}\mathbf{H}, T$, part (3) of the proposition implies that the set $\mathbf{H}$ contains G. So the set $\mathbf{G}$ includes the groups of all ambients that can be immediately crossed by $m[P]$, and the set $\mathbf{H}$ includes the groups of all ambients that can be immediately opened within $m[P]$.

A corollary of Theorem 6.1 is that these bounds on ambient behavior apply not just to ambients contained within Q, but to ambients contained in any process reachable by a series of reductions from Q.

For the sake of simplicity and brevity, our discussion in this section is fairly informal. In their recent work on a derivative of the ambient calculus, Bugliesi and Castagna [8] state a formal safety property induced by a type system for ambients. To do so, they introduce a precise notion of process residuals.

8. ENCODING A DISTRIBUTED LANGUAGE

Several typed and untyped distributed languages have been proposed [35, 22]. They come with notions of locations, agents, threads, mobility, and so on. Typed translations of procedural and object-oriented programming languages into formal type systems have been studied for several reasons including type soundness [2] and compilation optimisations [28]. In the same way, we aim to reduce the constructs of agent languages to appropriate type systems that capture their fundamental characteristics.

In this section, we consider a particular example, a fragment of a typed, distributed language in which mobile threads can migrate between immobile network nodes. We obtain a semantics for this form of thread mobility via a translation into the ambient calculus. In the translation, ambients model both threads and nodes. The encoding can be typed in all three of the systems presented in this paper; for the sake of brevity we describe the encoding only for the full system of Section 6. The encoding illustrates how groups can be used to partition the set of ambient names according to their intended usage, and how opening and crossing control allows the programmer to state interesting invariants. In particular, the typing of the translation guarantees that an ambient modeling a node moves neither subjectively nor objectively. On the other hand, an ambient modeling a thread is free to move subjectively, but is guaranteed not to move objectively.

8.1. The Distributed Language

The computational model is that there is an unstructured collection of named network nodes, each of which hosts a collection of named communication channels and anonymous threads. This is similar to the computational models underlying various distributed variants of the π-calculus, such as those proposed by Amadio and Prasad [4], Riely and Hennessy [32], and Sewell [33]. In an earlier paper [12], we showed how to mimic Telescript's computational model by translation into the ambient calculus. In the language fragment we describe here, communication is based on named communication channels (as in the π-calculus) rather than by direct agent-to-agent communication (as in our stripped down version of Telescript). As in our previous paper, we focus on language constructs for mobility, synchronization, and communication. We omit standard constructs for data processing and control flow. They could easily be added.

To introduce the syntax of our language fragment, here is a simple example:

$$\begin{array}{l} \mathit{node}\ a\,[\mathit{channel}\ a_c \mid \mathit{thread}[\overline{a_c}\langle b, b_c\rangle]] \mid \mathit{node}\ b\,[\mathit{channel}\ b_c] \mid \\ \mathit{node}\ c\,[\mathit{thread}[\mathit{go}\ a.a_c(x{:}\mathit{Node}, y{:}\mathit{Ch}[\mathit{Node}]).\mathit{go}\ x.\overline{y}\langle a\rangle] \end{array}$$

This program describes a network consisting of three network nodes, named a, b, and c. Node a hosts a channel a_c and a thread running the code $\overline{a_c}\langle b, b_c\rangle$, which simply sends the pair $\langle b, b_c\rangle$ on the channel a_c. Node b hosts a channel b_c. Finally, node c hosts a single thread, running the code:

$$\mathit{go}\ a.a_c(x{:}\mathit{Node}, y{:}\mathit{Ch}[\mathit{Node}]).\mathit{go}\ x.\overline{y}\langle a\rangle$$

The effect of this is to move the thread from node c to node a. There it awaits a message sent on the communication channel a_c. We may assume that it receives the message $\langle b, b_c\rangle$ being sent by the thread already at a. (If there were another thread at node a sending another message, the receiver thread would end up receiving one or other of the messages.) The thread then migrates to node b, where it transmits a message a on the channel b_c.

Messages on communication channels are assigned types, ranged over by Ty. The type Node is the type of names of network nodes. The type $\mathit{Ch}[\mathit{Ty}_1, \ldots, \mathit{Ty}_k]$ is the type of a polyadic communication channel. The messages communicated on such a channel are k-tuples whose components have types $\mathit{Ty}_1, \ldots, \mathit{Ty}_k$. In the setting of the example above, channel a_c has type $\mathit{Ch}[\mathit{Node}, \mathit{Ch}[\mathit{Node}]]$, and channel b_c has type $\mathit{Ch}[\mathit{Node}]$.

Next, we describe the formal grammar of our language fragment. A *network*, Net, is a collection of nodes, built up using composition $\mathit{Net} \mid \mathit{Net}$ and restrictions $(\nu n{:}\mathit{Ty})\mathit{Net}$. A *crowd*, Cro, is the group of threads and channels hosted by a node. Like networks, crowds are built up using composition $\mathit{Cro} \mid \mathit{Cro}$ and restriction $(\nu n{:}\mathit{Ty})\mathit{Cro}$. A *thread*, Th, is a mobile thread of control. As well as the constructs illustrated above, a thread may include the contructs $\mathit{fork}(\mathit{Cro}).\mathit{Th}$ and $\mathit{spawn}\ n\,[\mathit{Cro}].\mathit{Th}$. The first forks a new crowd Cro inside the current node, and continues with Th. The second spawns a new node $\mathit{node}\,n\,[\mathit{Cro}]$ outside the current node, at the network level, and continues with Th.

A Fragment of a Typed, Distributed Programming Language:

$\mathit{Ty} ::=$	type
$\quad \mathit{Node}$	name of a node
$\quad \mathit{Ch}[\mathit{Ty}_1, \ldots, \mathit{Ty}_k]$	name of a channel
$\mathit{Net} ::=$	network
$\quad (\nu n{:}\mathit{Ty})\mathit{Net}$	restriction
$\quad \mathit{Net} \mid \mathit{Net}$	network composition
$\quad \mathit{node}\ n\,[\mathit{Cro}]$	node
$\mathit{Cro} ::=$	crowd of channels and threads
$\quad (\nu n{:}\mathit{Ty})\mathit{Cro}$	restriction
$\quad \mathit{Cro} \mid \mathit{Cro}$	crowd composition

$\quad$ *channel* c	$\quad$ channel
$\quad$ *thread*$[Th]$	$\quad$ thread
Th ::=	thread
$\quad$ *go* $n.Th$	$\quad$ migration
$\quad$ $\overline{c}\langle n_1, \ldots, n_k \rangle$	$\quad$ output to a channel
$\quad$ $c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th$	$\quad$ input from a channel
$\quad$ *fork*$(Cro).Th$	$\quad$ fork a crowd
$\quad$ *spawn* $n\,[Cro].Th$	$\quad$ spawn a new node

In the phrases $(\nu n{:}Ty)Net$ and $(\nu n{:}Ty)Cro$, the name n is bound; its scope is Net and Cro, respectively. In the phrase $c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th$, the names x_1, $\ldots$, x_k are bound; their scope is the phrase Th.

The type system of our language controls the typing of messages on communication channels, much as in previous schemes for the π-calculus [27]. We formalize the type system as five judgments defined by the following rules.

Judgments:

$E \vdash \diamond$	good environment
$E \vdash n : Ty$	name n has type Ty
$E \vdash Net$	good network
$E \vdash Cro$	good crowd
$E \vdash Th$	good thread

Good Environment:

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

Name has Type:

$$\frac{E, n{:}Ty, E' \vdash \diamond}{E, n{:}Ty, E' \vdash n : Ty}$$

Good Network:

$$\frac{E, n{:}Ty \vdash Net}{E \vdash (\nu n{:}Ty)Net} \qquad \frac{E \vdash Net \quad E \vdash Net'}{E \vdash Net \mid Net'} \qquad \frac{E \vdash n : Node \quad E \vdash Cro}{E \vdash node\ n\,[Cro]}$$

Good Crowd:

$$\frac{E, n{:}Ty \vdash Cro}{E \vdash (\nu n{:}Ty)Cro} \qquad \frac{E \vdash Cro \quad E \vdash Cro'}{E \vdash Cro \mid Cro'}$$

$$\frac{E \vdash c : Ch[Ty_1, \ldots, Ty_k]}{E \vdash channel\ c} \qquad \frac{E \vdash Th}{E \vdash thread[Th]}$$

Good Thread:

$$\frac{E \vdash n : \mathit{Node} \quad E \vdash \mathit{Th}}{E \vdash \mathit{go}\, n.\mathit{Th}}$$

$$\frac{E \vdash c : \mathit{Ch}[\mathit{Ty}_1, \ldots, \mathit{Ty}_k] \quad E \vdash n_i : \mathit{Ty}_i \quad \forall i \in 1..k}{E \vdash \overline{c}\langle n_1, \ldots, n_k\rangle}$$

$$\frac{E \vdash c : \mathit{Ch}[\mathit{Ty}_1, \ldots, \mathit{Ty}_k] \quad E, x_1{:}\mathit{Ty}_1, \ldots, x_k{:}\mathit{Ty}_k \vdash \mathit{Th}}{E \vdash c(x_1{:}\mathit{Ty}_1, \ldots, x_k{:}\mathit{Ty}_k).\mathit{Th}}$$

$$\frac{E \vdash \mathit{Cro} \quad E \vdash \mathit{Th}}{E \vdash \mathit{fork}(\mathit{Cro}).\mathit{Th}} \qquad \frac{E \vdash n : \mathit{Node} \quad E \vdash \mathit{Cro} \quad E \vdash \mathit{Th}}{E \vdash \mathit{spawn}\, n\, [\mathit{Cro}].\mathit{Th}}$$

8.2. Typed Translation to the Ambient Calculus

In this section, we translate our distributed language to the typed ambient calculus of Section 6.

The basic idea of the translation is that ambients model nodes, channels, and threads. For each channel, there is a name for a buffer ambient, of group Ch^b, and there is a second name, of group Ch^p, for packets exchanged within the channel buffer. Similarly, for each node, there is a name, of group Node^b, for the node itself, and a second name, of group Node^p, for short-lived ambients that help fork crowds within the node, or to spawn other nodes. Finally, there is a group Thr to classify the names of ambients that model threads. The following table summarizes these five groups:

Global Groups Used in the Translation:

Node^b	ambients that model nodes
Node^p	ambients to help fork crowds or spawn nodes
Ch^b	ambients that model channel buffers
Ch^p	ambients that model packets on a channel
Thr	ambients that model threads

We begin the translation by giving types in the ambient calculus corresponding to types in the distributed language. Each type Ty gets translated to a pair $[\![\mathit{Ty}]\!]^b$, $[\![\mathit{Ty}]\!]^p$ of ambient calculus types. Throughout this section, we omit the curly braces when writing singleton group sets; for example, we write $\frown\mathit{Node}^b$ as a shorthand for $\frown\{\mathit{Node}^b\}$.

First, if Ty is a node type, $[\![\mathit{Ty}]\!]^b$ is the type of an ambient (of group Node^b) modeling a node, and $[\![\mathit{Ty}]\!]^p$ is the type of helper ambients (of group Node^p). Second, if Ty is a channel type, $[\![\mathit{Ty}]\!]^b$ is the type of an ambient (of group Ch^b) modeling a channel buffer, and $[\![\mathit{Ty}]\!]^p$ is the type of a packet ambient (of group Ch^p).

Translations $\llbracket Ty \rrbracket^b$, $\llbracket Ty \rrbracket^p$ of a Type Ty:

$\llbracket Node \rrbracket^b \triangleq Node^b \frown Node^b[\frown\{\}, {}^\circ Node^p, Shh]$
$\llbracket Node \rrbracket^p \triangleq Node^p \frown Thr[\frown\{\}, {}^\circ Node^p, Shh]$
$\llbracket Ch[Ty_1, \dots, Ty_k] \rrbracket^b \triangleq$
$\quad Ch^b \frown\{\}[\frown\{\}, {}^\circ Ch^p, \llbracket Ty_1 \rrbracket^b \times \llbracket Ty_1 \rrbracket^p \times \cdots \times \llbracket Ty_k \rrbracket^b \times \llbracket Ty_k \rrbracket^p]$
$\llbracket Ch[Ty_1, \dots, Ty_k] \rrbracket^p \triangleq$
$\quad Ch^p \frown\{Thr, Ch^b\}[\frown\{\}, {}^\circ Ch^p, \llbracket Ty_1 \rrbracket^b \times \llbracket Ty_1 \rrbracket^p \times \cdots \times \llbracket Ty_k \rrbracket^b \times \llbracket Ty_k \rrbracket^p]$

These typings say a lot about the rest of the translation, because of the presence of five different groups. Nodes and helpers are silent ambients, whereas tuples of ambient names are exchanged within both channel buffers and packets. None of these ambients is subjectively mobile. On the other hand, nodes may objectively cross nodes, helpers may objectively cross threads, buffers are objectively immobile, and packets objectively cross both threads and buffers. Finally, both nodes and helpers may open only helpers, and both buffers and packets may open only packets. (Actually, as discussed in Section 5.2, the ${}^\circ Ch^p$ annotation inside the type of a packet c^p of group Ch^p means that c^p can be opened, and similarly for helpers.)

Next, we translate networks to typed processes. A restriction of a single name is mapped to restrictions of a couple of names: either names for a node and helpers, if the name is a node, or names for a buffer and packets, if the name is a channel. A composition is simply translated to a composition. A network node n is translated to an ambient named n^b representing the node, containing a replicated *open* n^p, where n^p is the name of helper ambients for that node.

Translation $\llbracket Net \rrbracket$ of a Network Net:

$\llbracket (\nu n{:}Ty)Net \rrbracket \triangleq (\nu n^b{:}\llbracket Ty \rrbracket^b)(\nu n^p{:}\llbracket Ty \rrbracket^p)\llbracket Net \rrbracket$
$\llbracket Net \mid Net \rrbracket \triangleq \llbracket Net \rrbracket \mid \llbracket Net \rrbracket$
$\llbracket node\ n\ [Cro] \rrbracket \triangleq n^b[!open\ n^p \mid \llbracket Cro \rrbracket_n]$

The translation $\llbracket Cro \rrbracket_n$ of a crowd is indexed by the name n of the node in which the crowd is located. Restrictions and compositions in crowds are translated like their counterparts at the network level. A channel c is represented by a buffer ambient c^b of group Ch^b. It is initially empty but for a replicated *open* c^p, where c^p is the name, of group Ch^p, of packets on the channel. The replication allows inputs and outputs on the channel to meet and exchange messages.

An ambient of the following type models each thread:

$$Thr \frown\{\}[\frown Node^b, {}^\circ Sync, Shh]$$

From the type, we know that a thread ambient is silent, that it crosses node boundaries by subjective moves but crosses nothing by objective moves, and that it may only open ambients in the *Sync* group. Such ambients help synchronize parallel processes in thread constructs such as receiving on a channel. A fresh group named *Sync* is created by a $(\nu Sync)$ in the translation of each thread. The existence of a separate lexical scope for *Sync* in each thread implies there can be no accidental transmission between threads of the names of private synchronization ambients.

Translation $[\![Cro]\!]_n$ of a Crowd Cro Located at Node n:

$[\![(\nu m{:}Ty)\,Cro]\!]_n \triangleq (\nu m^b{:}[\![Ty]\!]^b)(\nu m^p{:}[\![Ty]\!]^p)[\![Cro]\!]_n$
$[\![Cro \mid Cro]\!]_n \triangleq [\![Cro]\!]_n \mid [\![Cro]\!]_n$
$[\![channel\ c]\!]_n \triangleq c^b[!open\ c^p]$
$[\![thread\ Th]\!]_n \triangleq (\nu Sync)(\nu t{:}Thr \frown \{\}[\frown Node^b, ^\circ Sync, Shh])t[[\![Th]\!]_n^t]$
for $t \notin \{n\} \cup \{m^p, m^b \mid m \text{ free in } Th\}$

The translation $[\![Th]\!]_n^t$ of a thread is indexed by the name t of the thread and by the name n of the node in which the thread is enclosed. Each thread t is given a different name (this constraint can be formalized in many different ways).

A migration $go\ m.Th$ is translated to subjective moves taking the thread t out of the current node n and into the target node m.

An output $\overline{c}\langle n_1, \ldots, n_k\rangle$ is translated to a packet ambient c^p that travels to the channel buffer c^b, where it is opened, and outputs a tuple of names.

An input $c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th$ is translated to a packet ambient c^p that travels to the channel buffer c^b, where it is opened, and inputs a tuple of names; the tuple is returned to the host thread t by way of a synchronization ambient s, that exits the buffer and then returns to the thread.

A fork $fork(Cro).Th$ is translated to a helper ambient n^p that exits the thread t and gets opened within the enclosing node n. This unleashes the crowd Cro and allows a synchronization ambient s to return to the thread t, where it triggers the continuation Th.

A spawn $spawn\ m\ [Cro].Th$ is translated to a helper ambient n^p that exits the thread t and gets opened within the enclosing node n^b. This unleashes an objective move $go(out\ n^b).m^b[!open\ m^p \mid [\![Cro]\!]_m]$ that travels out of the node to the top, network level, where it starts the fresh node $m^b[!open\ m^p \mid [\![Cro]\!]_m]$. Concurrently, a synchronization ambient s returns to the thread t, where it triggers the continuation Th.

Translation $[\![Th]\!]_n^t$ of a Thread Th Named t Located at Node n:

$[\![go\ m.Th]\!]_n^t \triangleq out\ n^b.in\ m^b.[\![Th]\!]_m^t$
$[\![\overline{c}\langle n_1, \ldots, n_k\rangle]\!]_n^t \triangleq go(out\ t.in\ c^b).c^p[\langle n_1^b, n_1^p, \ldots, n_k^b, n_k^p\rangle]$
$[\![c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th]\!]_n^t \triangleq$
$(\nu s{:}Sync \frown \{Thr, Ch^b\}[\frown Node^b, ^\circ Sync, Shh])$
$(go(out\ t.in\ c^b).$
$c^p[(x_1^b{:}[\![Ty_1]\!]^b, x_1^p{:}[\![Ty_1]\!]^p, \ldots, x_k^b{:}[\![Ty_k]\!]^b, x_k^p{:}[\![Ty_k]\!]^p).$
$go(out\ c^b.in\ t).s[open\ s.[\![Th]\!]_n^t]] \mid$
$open\ s.s[])$
for $s \notin \{t, c^b, c^p\} \cup fn([\![Th]\!]_n^t)$
$[\![fork(Cro).Th]\!]_n^t \triangleq$
$(\nu s{:}Sync \frown Thr[\frown Node^b, ^\circ Sync, Shh])$
$(go\ out\ t.n^p[go\ in\ t.s[] \mid [\![Cro]\!]_n] \mid open\ s.[\![Th]\!]_n^t)$
for $s \notin \{t, n^p\} \cup [\![Cro]\!]_n \cup [\![Th]\!]_n^t$

$$
\begin{array}{l}
[\![\mathit{spawn}\ m\ [\mathit{Cro}].\mathit{Th}]\!]^t_n \triangleq \\
\quad (\nu s{:}\mathit{Sync}\frown\mathit{Thr}[\frown\mathit{Node}^b,{}^\circ\mathit{Sync},\mathit{Shh}]) \\
\quad\quad (\mathit{go\ out}\ t.n^p[\mathit{go\ in}\ t.s[]\mid \mathit{go\ out}\ n^b.m^b[!\mathit{open}\ m^p \mid [\![\mathit{Cro}]\!]_m]] \mid \\
\quad\quad\ \ \mathit{open}\ s.[\![\mathit{Th}]\!]^t_n) \\
\quad \text{for } s\notin\{t,n^b,n^p,m^b,m^p\}\cup \mathit{fn}([\![\mathit{Cro}]\!]_m)\cup\mathit{fn}([\![\mathit{Th}]\!]^t_n)
\end{array}
$$

Finally, we translate typing environments as follows.

Translation $[\![E]\!]$ of an Environment E:

$$
\begin{array}{rcl}
[\![\varnothing]\!] & \triangleq & \mathit{Node}^b,\mathit{Node}^p,\mathit{Ch}^b,\mathit{Ch}^p,\mathit{Thr} \\
[\![E,c{:}\mathit{Ty}]\!] & \triangleq & [\![E]\!],c^b{:}[\![\mathit{Ty}]\!]^b,c^p{:}[\![\mathit{Ty}]\!]^p
\end{array}
$$

Our translation preserves typing judgments:

PROPOSITION 8.1.

(1)*If* $E\vdash \mathit{Net}$ *then* $[\![E]\!]\vdash[\![\mathit{Net}]\!]:\frown\{\},{}^\circ\{\},\mathit{Shh}$.
(2)*If* $E\vdash\mathit{Cro}$ *and* $E\vdash n:\mathit{Node}$ *then* $[\![E]\!]\vdash[\![\mathit{Cro}]\!]_n:\frown\{\},{}^\circ\{\},\mathit{Shh}$.
(3)*If* $E\vdash \mathit{Th}$, $E\vdash n:\mathit{Node}$, $t\notin \mathit{dom}(E)$ *then*

$$[\![E]\!],\mathit{Sync},t{:}\mathit{Thr}\frown\{\}[\frown\mathit{Node}^b,{}^\circ\mathit{Sync},\mathit{Shh}]\vdash[\![\mathit{Th}]\!]^t_n:\frown\mathit{Node}^b,{}^\circ\mathit{Sync},\mathit{Shh}\ .$$

Proof. By inductions on derivations. ■

Apart from having more refined types, this translation is the same as a translation to the type system with binary annotations of [9]. (We discussed the same binary system in Section 6.3.) The translation shows that ambients can model a variety of concepts arising in mobile computation: nodes, threads, communication packets and buffers. Groups admit more precise typings for this translation than were possible in the system with binary annotations. For example, here we can tell that a thread ambient subjectively crosses only node ambients, but never crosses helpers, buffers, or packets, and that it is objectively immobile; in the binary system, all we can say is that a thread ambient was subjectively mobile and objectively immobile.

9. CONCLUSIONS

Our contribution is a type system for tracking the behavior of mobile computations. The system tracks the communication, mobility, and opening behavior of ambients, which are classified by *groups*. A group represents a collection of ambient names; ambient names belong to groups in the same sense that values belong to types. We studied the properties of a new process operator $(\nu G)P$ that lexically scopes groups. Using groups, our type system can impose behavioral constraints like "this ambient crosses only ambients in one set of groups, and only dissolves ambients in another set of groups". Although we have not implemented our type system, we assessed its expressiveness by encoding a distributed language featuring mobility of threads between network nodes. The encoding shows the usefulness of the type system in expressing properties of simple protocols for thread mobility.

Our ambient calculus is related to earlier distributed variants of the π-calculus, some of which have been equipped with type systems. The type system of Amadio [3] prevents a channel from being defined at more than one location. Sewell's system [33] tracks whether communications are local or non-local, so as to allow efficient implementation of local communication. In Riely and Hennessy's calculus [32], processes need appropriate permissions to perform actions such as migration; a well-typed process is guaranteed to possess the appropriate permission for any action it attempts. Other work on typing for mobile agents includes a type system by De Nicola, Ferrari, and Pugliese [16] that tracks the access rights an agent enjoys at different localities; type-checking ensures that an agent complies with its access rights.

Our groups are similar to the *sorts* used as static classifications of names in the π-calculus [27]. Our basic system of Section 4 is comparable to Milner's sort system for π, except that sorts in the π-calculus are mutually recursive; we would have to add a recursion operator to achieve a similar effect. Another difference is that an operator for sort creation does not seem to have been considered in the π-calculus literature. Our operator for group creation can guarantee secrecy properties, as we show in the setting of a typed π-calculus equipped with groups [11]. Our systems of Sections 5 and 6 depend on groups to constrain the opening and crossing behavior of processes. We are not aware of any uses of Milner's sorts to control process behavior beyond controlling the sorts of communicated names.

Apart from Milner's sorts, other static classifications of names occur in derivatives of the π-calculus. We mention two examples. In the type system of Abadi [1] for the spi calculus, names are classified by three static *security levels—Public*, *Secret*, and *Any*—to prevent insecure information flows. In the flow analysis of Bodei, Degano, Nielson, and Nielson [6] for the π-calculus, names are classified by static *channels* and *binders*, again with the purpose of establishing security properties. Although there is a similarity between these notions and groups, and indeed to sorts, nothing akin to our (νG) operator appears to have been studied.

There is a connection between groups and the region variables in the work of Tofte and Talpin [34] on region-based implementation of the λ-calculus. The store is split into a set of stack-allocated regions, and the type of each stored value is labelled with the region in which the value is stored. The scoping construct *letregion* ρ *in* e allocates a fresh region, binds it to the region variable ρ, evaluates e, and on completion, deallocates the region bound to ρ. The constructs *letregion* ρ *in* e and $(\nu G)P$ are similar in that they confer static scopes on the region variable ρ and the group G, respectively. One difference is that in our operational semantics $(\nu G)P$ is simply a scoping construct; it allocates no storage. Another is that scope extrusion laws do not seem to have been explicitly investigated for *letregion*. Still, we can interpret *letregion* in terms of (νG), as is reported elsewhere [15].

As noted in the introduction, the type systems presented in this article were first reported in conference papers on exchange types [12], mobility types [9], and ambient groups [10]. We conclude the article with a survey of other static analyses for the ambient calculus.

- Several papers examine the problem of computing safe approximations to the hierarchical structure of ambients, that is, of determining an approximation to the sets of ambients that may occur as children of other ambients. Nielson, Nielson, Hansen, and Jensen [29] present the first control flow analysis to address this problem. They present an algorithm for validating firewalls programmed in the ambient calculus. In subsequent work, Nielson and Nielson [31] and Nielson, Nielson, and Sagiv [30] present more accurate but also more expensive algorithms based, respectively, on regular tree grammars and on an interpretation in Kleene's three-valued logic.

• Abstract interpretation is a methodology for deriving program analyses systematically from the semantics of a programming language. Hansen, Jensen, Nielson, and Nielson [20] describe a constraint-based framework for abstract interpretation of mobile ambients; instances of the framework include an analysis counting occurrences of ambients, and also the original control flow analysis for the ambient calculus [29]. Levi and Maffeis [24] and Feret [19] present abstract interpretations based on alternative semantics of the ambient calculus.

• Some analyses have been developed in the setting of Levi and Sangiorgi's calculus of safe ambients [25], a generalization of the original ambient calculus that gives processes greater control over synchronization, and hence avoids certain kinds of nondeterminism. In their paper, Levi and Sangiorgi propose a type system to guarantee immobility and single-threadedness.

• Security properties are considered by several authors. Bugliesi and Castagna [8] describe a type system for safe ambients that checks security properties, including security in a distributed setting. They rely on a notion of ambient domain that is similar to the notion of an ambient group, but have no counterpart to the group creation operator. Dezani-Ciancaglini and Salvo [18] present a type system for safe ambients where each ambient has a security level, akin to a group. Unlike our system, security levels are partially ordered, allowing the system to express trust relationships. Degano, Levi, and Bodei [17, 23] refine Nielson and Nielson's original flow analysis [29] for the calculus of safe ambients. The analysis allows the proof of simple secrecy properties; they formally distinguish between trustworthy and untrustworthy ambients, and show that no trustworthy ambient may be opened immediately inside an untrustworthy ambient.

• Finally, Amtoft, Kfoury, and Pericas-Geertsen [5] propose a polymorphic ambient calculus, a conservative extension of our system of exchange types [12].

ACKNOWLEDGMENT

Silvano Dal Zilio commented on a draft of this paper. Ghelli acknowledges the support of Microsoft Research during the writing of this paper. The same author has also been partially supported by grants from the E.U. workgroup APPSEM, and by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica", project DATA-X. Comments from the anonymous referees were invaluable.

APPENDIX A

Proof of Subject Reduction

In this appendix we prove Theorem 6.1, the subject reduction property for the type system of Section 6, the richest of the three type systems presented in this paper. Proofs of subject reduction for the other two systems can be obtained as simplifications of this appendix.

We begin by stating some basic properties of the type system. The lemmas we state without proof can be proved by straightforward inductions on derivations. We use the notation $E \vdash \mathcal{J}$ to stand for an instance of any of the five different judgments of the system. We write $fn(\mathcal{J})$ and $fg(\mathcal{J})$ to stand for the names and groups, respectively, that occur free in $\mathcal{J}$. Moreover, if $\mathbf{G} = \{G_1, \ldots, G_k\}$ we write the notation $\mathbf{G}, E \vdash \mathcal{J}$ as a shorthand for $G_1, \ldots, G_k, E \vdash \mathcal{J}$.

LEMMA A.1. *If $E, E' \vdash \mathcal{J}$ then $E \vdash \diamond$ and $dom(E) \cap dom(E') = \varnothing$.*

Proof. The proof is by induction on the depth of the derivation of $E, E' \vdash \mathcal{J}$. ∎

LEMMA A.2. *If $E', n{:}W, E'' \vdash \mathcal{J}$ then $E' \vdash W$.*

Proof. By Lemma A.1 we have $E', n{:}W, E'' \vdash \mathcal{J} \;\Rightarrow\; E', n{:}W \vdash \diamond$, which must have been derived from $E' \vdash W$. ∎

We have two weakening lemmas:

LEMMA A.3. *If $E', E'' \vdash \mathcal{J}$ and $n \notin dom(E', E'')$ and $E' \vdash W$ then $E', n{:}W, E'' \vdash \mathcal{J}$.*

LEMMA A.4. *If $E', E'' \vdash \mathcal{J}$ and $G \notin dom(E', E'')$ then $E', G, E'' \vdash \mathcal{J}$.*

LEMMA A.5. *If $E \vdash M : W$ then $E \vdash W$.*

Proof. By induction on the derivation of $E \vdash M : W$, using Lemma A.2, Lemma A.3, and Lemma A.4 in case (Exp n). ∎

We state a useful corollary:

LEMMA A.6. *If $E', E'' \vdash \mathcal{J}$ and $E', \mathbf{G}, E'' \vdash \mathcal{J}'$ then $E', \mathbf{G}, E'' \vdash \mathcal{J}$.*

Proof. By Lemma A.1, $E', \mathbf{G}, E'' \vdash \mathcal{J}'$ implies that $\mathbf{G} \cap dom(E', E'') = \varnothing$. By Lemma A.4, this implies that $E', \mathbf{G}, E'' \vdash \mathcal{J}$. ∎

LEMMA A.7. *If $E \vdash n : W$ and $E \vdash n : W'$ then $W = W'$.*

Proof. Use Lemma A.1. ∎

LEMMA A.8. *If $E \vdash \mathcal{J}$ then $fn(\mathcal{J}) \subseteq dom(E)$ and $fg(\mathcal{J}) \subseteq dom(E)$.*

Proof. By induction on the derivation of $E \vdash \mathcal{J}$. ∎

LEMMA A.9. *If $E \vdash \diamond$ and $fg(W) \subseteq dom(E)$ then $E \vdash W$; if $E \vdash \diamond$ and $fg(F) \subseteq dom(E)$ then $E \vdash F$.*

Proof. By mutual induction on the structure of W and F. ■

Hereafter, let $fg(E'')$ be the set of all groups that occur either in the domain of E'' or in types occurring in E''.

LEMMA A.10. *If $E \vdash M : G \frown \mathbf{G}[F]$ than $M = n$ for some n.*

Proof. (Exp n) is the only rule that can derive $E \vdash M : G \frown \mathbf{G}[F]$. ■

LEMMA A.11. *If $E', G, E'' \vdash n : W$ and $G \in fg(W)$ then $G \in fg(E'')$.*

Proof. If n were defined in E', then, by Lemmas A.2 and A.8, we would have $G \in dom(E')$, which contradicts Lemma A.1. Hence n is defined in E'', and the thesis follows by Lemma A.7. ■

We have two strengthening lemmas.

LEMMA A.12. *If $E', n{:}W, E'' \vdash \mathcal{J}$ and $n \notin fn(\mathcal{J})$ then $E', E'' \vdash \mathcal{J}$.*

LEMMA A.13. *If $E', G, E'' \vdash \mathcal{J}$ and $G \notin fg(\mathcal{J}) \cup fg(E'')$ then $E', E'' \vdash \mathcal{J}$.*

Proof. By induction on the derivation of $E', G, E'' \vdash \mathcal{J}$. Needs Lemma A.11 in cases (Exp In), (Exp Out), (Exp Open), and (Proc Amb). Lemma A.10 is also used for (Proc Amb) and (Proc Go). ■

LEMMA A.14. *If $E \vdash P : F$ then $E \vdash F$.*

Proof. By induction on the derivation of $E \vdash P : F$. Needs Lemmas A.12 and A.13 in cases (Proc Res), (Proc GRes), and (Proc Input) and Lemma A.5 in case (Proc Output). ■

Next, we have four exchange lemmas. They are all proved by induction on the derivation, exploiting the weakening and strengthening lemmas in the crucial cases (Env n) and (Env G).

LEMMA A.15. *If $E', n{:}W', m{:}W'', E'' \vdash \mathcal{J}$ then $E', m{:}W'', n{:}W', E'' \vdash \mathcal{J}$.*

LEMMA A.16. *If $E', n{:}W', G, E'' \vdash \mathcal{J}$ then $E', G, n{:}W', E'' \vdash \mathcal{J}$.*

LEMMA A.17. *If $E', G, n{:}W', E'' \vdash \mathcal{J}$ and $G \notin fg(W')$ then $E', n{:}W', G, E'' \vdash \mathcal{J}$.*

LEMMA A.18. *If $E', G, H, E'' \vdash \mathcal{J}$ then $E', H, G, E'' \vdash \mathcal{J}$.*

We have a substitution lemma:

LEMMA A.19. *If $E', n{:}W, E'' \vdash \mathcal{J}$ and $E' \vdash M : W$ then $E', E'' \vdash \mathcal{J}\{n \leftarrow M\}$.*

Proof. By induction on the derivation of $E', n{:}W, E'' \vdash \mathcal{J}$. Most cases are straightforward, with the exception of (Exp n), (Exp In), (Exp Out), and (Exp Open), when the name that appears in the rule is exactly n. For the case (Exp n), we get the desired judgment $E', E'' \vdash M : W$ from $E' \vdash M : W$ by the weakening lemmas, Lemmas A.3 and A.4. For the cases (Exp In), (Exp Out), and (Exp Open), we use Lemma A.10 to show that M is actually a name m. By the weakening lemmas, we get $E', E'' \vdash m : W$, and then may draw the desired conclusion with (Exp In), (Exp Out), or (Exp Open), respectively. ■

Next, we prove that structural congruence preserves typing judgments, possibly with the inclusion of fresh group names.

PROPOSITION A.1. *If $E \vdash P : F$ and $P \equiv Q$ then there are groups $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : F$.*

Proof. The proposition follows by showing that $P \equiv Q$ implies:

(1) If $E \vdash P : F$ then $\exists \mathbf{G}$ such that $\mathbf{G}, E \vdash Q : F$.
(2) If $E \vdash Q : F$ then $\exists \mathbf{G}$ such that $\mathbf{G}, E \vdash P : F$.

We proceed by induction on the derivation of $P \equiv Q$.

(Struct Refl) Trivial.

(Struct Symm) Then $Q \equiv P$. For (1), assume $E \vdash P : F$. By induction hypothesis (2), $Q \equiv P$ implies that $\exists \mathbf{G}$ such that $\mathbf{G}, E \vdash Q : F$. Part (2) is symmetric.

(Struct Trans) Then $P \equiv R$, $R \equiv Q$ for some R. For (1), assume $E \vdash P : F$. By induction hypothesis (1), $\exists \mathbf{G}.\ \mathbf{G}, E \vdash R : F$. Again by induction hypothesis (1), $\exists \mathbf{H}.\ \mathbf{H}, \mathbf{G}, E \vdash Q : F$. Part (2) is symmetric.

(Struct Res) Then $P = (\nu n{:}W)P'$ and $Q = (\nu n{:}W)Q'$, with $P' \equiv Q'$. For (1), assume $E \vdash P : F$. This must have been derived from (Proc Res), with $E, n{:}G \frown \mathbf{G}'[F'] \vdash P' : F$, where $W = G \frown \mathbf{G}'[F']$. By induction hypothesis (1), $\exists \mathbf{G}.\ \mathbf{G}, E, n{:}G \frown \mathbf{G}'[F'] \vdash Q' : F$. By (Proc Res), $\mathbf{G}, E \vdash (\nu n{:}W)Q' : F$. Part (2) is symmetric.

(Struct GRes) Then $P = (\nu G)P'$ and $Q = (\nu G)Q'$, with $P' \equiv Q'$. For (1), assume $E \vdash P : F$. This must have been derived from (Proc GRes), with $E, G \vdash P' : F$ where $G \notin fg(F)$. By induction hypothesis (1), $\exists \mathbf{G}.\ \mathbf{G}, E, G \vdash Q' : F$. By (Proc GRes), $\mathbf{G}, E \vdash (\nu G)Q' : F$. Part (2) is symmetric.

(Struct Par) Then $P = P' \mid R$, $Q = Q' \mid R$, and $P' \equiv Q'$. For (1), assume $E \vdash P' \mid R : F$. This must have been derived from (Proc Par), with $E \vdash P' : F$, $E \vdash R : F$. By induction hypothesis (1), $\exists \mathbf{G}.\ \mathbf{G}, E \vdash Q' : F$. By Lemma A.6, $\mathbf{G}, E \vdash R : F$. By (Proc Par), $\mathbf{G}, E \vdash Q' \mid R : F$. Part (2) is symmetric.

(Struct Repl) Then $P = !P'$, $Q = !Q'$, and $P' \equiv Q'$. For (1), assume $E \vdash P : F$. This must have been derived from (Proc Repl), with $E \vdash P' : F$. By induction

hypothesis (1), $\exists \mathbf{G}.\ \mathbf{G}, E \vdash Q' : F$. By (Proc Repl), $\mathbf{G}, E \vdash !Q' : F$. Part (2) is symmetric.

(Struct Amb) Then $P = M[P']$, $Q = M[Q']$, and $P' \equiv Q'$. For (1), assume $E \vdash P : F$. This must have been derived from (Proc Amb), with $E \vdash F$, $E \vdash M : G \frown \mathbf{G}'[F']$ and $E \vdash P' : F'$, for some $G, F', \mathbf{G}'$. By induction hypothesis (1), $\exists \mathbf{G}.\ \mathbf{G}, E \vdash Q' : F'$. By Lemma A.6, $\mathbf{G}, E \vdash F$ and $\mathbf{G}, E \vdash M : G \frown \mathbf{G}'[F']$. By (Proc Amb), $\mathbf{G}, E \vdash M[Q'] : F$. Part (2) is symmetric.

(Struct Action) Then $P = M.P'$, $Q = M.Q'$, and $P' \equiv Q'$. For (1), assume $E \vdash P : F$. This must have been derived from (Proc Action), with $E \vdash M : \mathit{Cap}[F]$ and $E \vdash P' : F$. By induction hypothesis (1), $\exists \mathbf{G}.\ \mathbf{G}, E \vdash Q' : F$. By Lemma A.6, $\mathbf{G}, E \vdash M : \mathit{Cap}[F]$. By (Proc Action), $\mathbf{G}, E \vdash M.Q' : F$.

Part (2) is symmetric.

(Struct Input) Then $P = (n_1{:}W_1, \ldots, n_k{:}W_k).P'$, $Q = (n_1{:}W_1, \ldots, n_k{:}W_k).Q'$, and $P' \equiv Q'$. For (1), assume $E \vdash P : F$. This must have been derived from (Proc Input), with $E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P' : F$, where $F = {\frown}\mathbf{G}', {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k$. By induction hypothesis, $\exists \mathbf{G}.\ \mathbf{G}, E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash Q' : F$. By (Proc Input), $\mathbf{G}, E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).Q' : F$. Part (2) is symmetric.

(Struct Go) Then $P = \mathit{go}\ N.M[P']$, $Q = \mathit{go}\ N.M[Q']$, and $P' \equiv Q'$. For (1), assume $E \vdash P : F$. This must have been derived from (Proc Go), with $E \vdash F$, $E \vdash N : \mathit{Cap}[F'']$, $E \vdash M : G \frown \mathbf{G}'[F']$ and $E \vdash P' : F'$, with $F'' = {\frown}\mathbf{G}', {}^{\circ}\{\}, \mathit{Shh}$, for some G, $\mathbf{G}'$, F'. By induction hypothesis (1), $\exists \mathbf{G}.\ \mathbf{G}, E \vdash Q' : F'$. By Lemma A.6, $\mathbf{G}, E \vdash F$ and $\mathbf{G}, E \vdash N : \mathit{Cap}[F'']$ and $\mathbf{G}, E \vdash M : G \frown \mathbf{G}'[F']$. By (Proc Go), $\mathbf{G}, E \vdash \mathit{go}\ N.M[Q'] : F$. Part (2) is symmetric.

(Struct Par Comm) Then $P = P' \mid P''$ and $Q = P'' \mid P'$.

For (1), assume $E \vdash P' \mid P'' : F$. This must have been derived from $E \vdash P' : F$ and $E \vdash P'' : F$. By (Proc Par), $E \vdash P'' \mid P' : F$. Hence, $E \vdash Q : F$.

Part (2) is symmetric.

(Struct Par Assoc) Then $P = (P' \mid P'') \mid P'''$ and $Q = P' \mid (P'' \mid P''')$.

For (1), assume $E \vdash (P' \mid P'') \mid P''' : F$. This must have been derived from (Proc Par) twice, with $E \vdash P' : F$, $E \vdash P'' : F$, and $E \vdash P''' : F$. By (Proc Par) twice, $E \vdash P' \mid (P'' \mid P''') : F$. Hence $E \vdash Q : F$.

Part (2) is symmetric.

(Struct Repl Par) Then $P = !P'$ and $Q = P' \mid !P'$.

For (1), assume $E \vdash !P' : F$. This must have been derived from (Proc Repl), with $E \vdash P' : F$. By (Proc Par), $E \vdash P' \mid !P' : F$. Hence, $E \vdash Q : F$.

For (2), assume $E \vdash P' \mid !P' : F$. This must have been derived from (Proc Par), with $E \vdash P' : F$ and $E \vdash !P' : F$. Hence, $E \vdash P : F$.

(Struct Res Res) Then $P = (\nu n_1{:}W_1)(\nu n_2{:}W_2)P'$ and $Q = (\nu n_2{:}W_2)(\nu n_1{:}W_1)P'$ with $n_1 \neq n_2$. For (1), assume $E \vdash (\nu n_1{:}W_1)(\nu n_2{:}W_2)P' : F$. This must have been derived from (Proc Res) twice, with $E, n_1{:}G_1 \frown \mathbf{G}_1[F_1], n_2{:}G_2 \frown \mathbf{G}_2[F_2] \vdash P' : F$, where $W_1 = G_1 \frown \mathbf{G}_1[F_1]$ and $W_2 = G_2 \frown \mathbf{G}_2[F_2]$. By Lemma A.15, we have $E, n_2{:}G_2 \frown \mathbf{G}_2[F_2], n_1{:}G_1 \frown \mathbf{G}_1[F_1] \vdash P' : F$. By (Proc Res) twice we have $E \vdash (\nu n_2{:}W_2)(\nu n_1{:}W_1)P' : F$. Part (2) is symmetric.

(Struct Res Par) Then $P = (\nu n{:}W)(P' \mid P'')$ and $Q = P' \mid (\nu n{:}W)P''$, with $n \notin fn(P')$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc Res), with $E, n{:}G \frown \mathbf{G}[F'] \vdash P' \mid P'' : F$ and $W = G \frown \mathbf{G}[F']$, and from (Proc Par), with $E, n{:}G \frown \mathbf{G}[F'] \vdash P' : F$ and $E, n{:}G \frown \mathbf{G}[F'] \vdash P'' : F$. By Lemma A.12, since $n \notin fn(P')$, we have $E \vdash P' : F$. By (Proc Res) we have $E \vdash (\nu n{:}G \frown \mathbf{G}[F'])P'' : F$. By (Proc Par) we have $E \vdash P' \mid (\nu n{:}G \frown \mathbf{G}[F'])P'' : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash Q : F$. This must have been derived from (Proc Par), with $E \vdash P' : F$ and $E \vdash (\nu n{:}W)P'' : F$, and from (Proc Res), with $E, n{:}G \frown \mathbf{G}[F'] \vdash P'' : F$ and $W = G \frown \mathbf{G}[F']$. By Lemma A.1, $n \notin dom(E)$. By Lemma A.2, $E \vdash G \frown \mathbf{G}[F']$. By Lemma A.3, $E, n{:}G \frown \mathbf{G}[F'] \vdash P' : F$. By (Proc Par), $E, n{:}G \frown \mathbf{G}[F'] \vdash P' \mid P'' : F$. By (Proc Res), $E \vdash (\nu n{:}G \frown \mathbf{G}[F'])(P' \mid P'') : F$, that is, $E \vdash P : F$.

(Struct Res Amb) Then $P = (\nu n{:}W)m[P']$ and $Q = m[(\nu n{:}W)P']$, with $n \neq m$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc Res) with $E, n{:}G \frown \mathbf{G}[F'] \vdash m[P'] : F$ with $W = G \frown \mathbf{G}[F']$, and from (Proc Amb) with $E, n{:}G \frown \mathbf{G}[F'] \vdash F$ and $E, n{:}G \frown \mathbf{G}[F'] \vdash m : H \frown \mathbf{G}'[F'']$ and $E, n{:}G \frown \mathbf{G}[F'] \vdash P' : F''$ for some H, F'', $\mathbf{G}'$. By (Proc Res) we have $E \vdash (\nu n{:}G \frown \mathbf{G}[F'])P' : F''$. By Lemma A.12, $E \vdash F$, and $E \vdash m : H \frown \mathbf{G}'[F'']$ (by $n \neq m$). By (Proc Amb), $E \vdash m[(\nu n{:}G \frown \mathbf{G}[F'])P'] : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash Q : F$. This must have been derived from (Proc Amb) with $E \vdash F$, $E \vdash m : H \frown \mathbf{G}'[F'']$ and $E \vdash (\nu n{:}W)P' : F''$, and from (Proc Res), with $E, n{:}G \frown \mathbf{G}[F'] \vdash P' : F''$ and $W = G \frown \mathbf{G}[F']$. By Lemma A.1, $n \notin dom(E)$. By Lemma A.2, $E, n{:}G \frown \mathbf{G}[F'] \vdash P' : F''$ implies $E \vdash G \frown \mathbf{G}[F']$. By Lemma A.3, $E, n{:}G \frown \mathbf{G}[F'] \vdash F$ and $E, n{:}G \frown \mathbf{G}[F'] \vdash m : H \frown \mathbf{G}'[F'']$. By (Proc Amb), $E, n{:}G \frown \mathbf{G}[F'] \vdash m[P'] : F$. By (Proc Res), $E \vdash (\nu n{:}G \frown \mathbf{G}[F'])m[P'] : F$, that is, $E \vdash P : F$.

(Struct GRes Res) Then $P = (\nu G)(\nu n{:}W)P'$ and $Q = (\nu n{:}W)(\nu G)P'$ with $G \notin fg(W)$.

For (1), assume $E \vdash (\nu G)(\nu n{:}W)P' : F$. This must have been derived from (Proc GRes), with $E, G \vdash (\nu n{:}W)P' : F$ and $G \notin fg(F)$, and from (Proc Res), with $E, G, n{:}G' \frown \mathbf{G}[F'] \vdash P' : F$, where $W = G' \frown \mathbf{G}[F']$. Since $G \notin fg(W)$ by hypothesis, by Lemma A.17 we have $E, n{:}G' \frown \mathbf{G}[F'], G \vdash P' : F$. We know that $G \notin fg(F)$, hence by (Proc GRes) we have $E, n{:}G' \frown \mathbf{G}[F'] \vdash (\nu G)P' : F$. Finally from (Proc Res) we have $E \vdash (\nu n{:}W)(\nu G)P' : F$.

For (2), assume $E \vdash (\nu n{:}W)(\nu G)P' : F$. This must have been derived from (Proc Res), with $E, n{:}G' \frown \mathbf{G}[F'] \vdash (\nu G)P' : F$, where $W = G' \frown \mathbf{G}[F']$, and from (Proc GRes), with $E, n{:}G' \frown \mathbf{G}[F'], G \vdash P' : F$, with $G \notin fg(F)$. By Lemma A.16, $E, G, n{:}G' \frown \mathbf{G}[F'] \vdash P' : F$. The thesis follows by applying (Proc Res) and (Proc GRes).

(Struct GRes GRes) Then $P = (\nu G_1)(\nu G_2)P'$ and $Q = (\nu G_2)(\nu G_1)P'$.

For (1), assume $E \vdash (\nu G_1)(\nu G_2)P' : F$. This must have been derived from (Proc GRes) twice, with $E, G_1, G_2 \vdash P' : F$ and $G_2 \notin fg(F)$, $G_1 \notin fg(F)$. By Lemma A.18 we have $E, G_2, G_1 \vdash P' : F$. By (Proc Res) twice we have $E \vdash (\nu G_2)(\nu G_1)P' : F$.

Part (2) is symmetric.

(Struct GRes Par) Then $P = (\nu G)(P' \mid P'')$ and $Q = P' \mid (\nu G)P''$, with $G \notin fg(P')$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc GRes), with $E, G \vdash P' \mid P'' : F$ and $G \notin fg(F)$, and from (Proc Par), with $E, G \vdash P' : F$ and $E, G \vdash P'' : F$. By Lemma A.13, since $G \notin fg(P') \cup fg(F)$, we have $E \vdash P' : F$. By (Proc GRes) we have $E \vdash (\nu G)P'' : F$. By (Proc Par) we have $E \vdash P' \mid (\nu G)P'' : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash Q : F$. This must have been derived from (Proc Par), with $E \vdash P' : F$ and $E \vdash (\nu G)P'' : F$, and from (Proc GRes), with $E, G \vdash P'' : F$ and $G \notin fg(F)$. By Lemma A.1, $G \notin dom(E)$. By Lemma A.4, $E, G \vdash P' : F$. By (Proc Par), $E, G \vdash P' \mid P'' : F$. By (Proc GRes), since $G \notin fg(F)$, $E \vdash (\nu G)(P' \mid P'')$, that is, $E \vdash P : F$.

(Struct GRes Amb) Then $P = (\nu G)m[P']$ and $Q = m[(\nu G)P']$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc GRes) with $E, G \vdash m[P'] : F$ with $G \notin fg(F)$, and from (Proc Amb) with $E, G \vdash F$, $E, G \vdash m : G' \frown \mathbf{G}[F']$, and $E, G \vdash P' : F'$ for some G', $\mathbf{G}$, F'. By Lemma A.13, $E, G \vdash F$ and $G \notin fg(F)$ imply $E \vdash F$. The judgment $E, G \vdash m : G' \frown \mathbf{G}[F']$ must have been derived from (Exp n), hence $m \in dom(E)$. Hence, by (Exp n) and by Lemma A.7, $E \vdash m : G' \frown \mathbf{G}[F']$. By Lemma A.1, $G \notin dom(E)$. Hence, by $E \vdash m : G' \frown \mathbf{G}[F']$ and Lemma A.8, $G \notin fg(G' \frown \mathbf{G}[F'])$ and so $G \notin fg(F')$. By (Proc GRes) we have $E \vdash (\nu G)P' : F'$. By (Proc Amb), $E \vdash m[(\nu G)P'] : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash Q : F$. This must have been derived from (Proc Amb) with $E \vdash F$, $E \vdash m : G' \frown \mathbf{G}[F']$, and $E \vdash (\nu G)P' : F'$, for some G', $\mathbf{G}$, F', and from (Proc GRes), with $E, G \vdash P' : F'$ and $G \notin fg(F')$. By Lemma A.1, $G \notin dom(E)$. By Lemma A.4, $E, G \vdash m : G' \frown \mathbf{G}[F']$ and $E, G \vdash F$. By (Proc Amb), $E, G \vdash m[P'] : F$. By Lemma A.8, $E \vdash F$ and $G \notin dom(E)$ imply $G \notin fg(F)$. By (Proc GRes), $E \vdash (\nu G)m[P'] : F$, that is, $E \vdash P : F$.

(Struct Zero Par) Then $P = P' \mid \mathbf{0}$ and $Q = P'$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc Par) with $E \vdash P' : F$ and $E \vdash \mathbf{0} : F$. Hence, $E \vdash Q : F$.

For (2), assume $E \vdash P' : F$. By Lemma A.14, $E \vdash F$. By (Proc Zero), $E \vdash \mathbf{0} : F$. By (Proc Par), $E \vdash P' \mid \mathbf{0} : F$, that is, $E \vdash P : F$.

(Struct Zero Res) Then $P = (\nu n{:}G \frown \mathbf{G}'[F'])\mathbf{0}$ and $Q = \mathbf{0}$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc Res) with $E, n{:}G \frown \mathbf{G}'[F'] \vdash \mathbf{0} : F$. By Lemma A.12, $E \vdash \mathbf{0} : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash \mathbf{0} : F$. We identify processes up to consistent renaming of bound names and groups, hence we may assume that the bound name n does not occur in $dom(E)$. Let $\mathbf{G}$ be $fg(G \frown \mathbf{G}'[F']) - dom(E)$. By Lemma A.4, $\mathbf{G}, E \vdash \diamond$. By Lemma A.9, $\mathbf{G}, E \vdash G \frown \mathbf{G}'[F']$. By Lemma A.14, $E \vdash F$. By repeated application of Lemma A.4, $\mathbf{G}, E \vdash F$. By Lemma A.3, $\mathbf{G}, E, n{:}G \frown \mathbf{G}'[F'] \vdash F$. By (Proc Zero), $\mathbf{G}, E, n{:}G \frown \mathbf{G}'[F'] \vdash \mathbf{0} : F$. By (Proc Res), $\mathbf{G}, E \vdash (\nu n{:}G \frown \mathbf{G}'[F'])\mathbf{0} : F$, that is, $\mathbf{G}, E \vdash P : F$.

(Struct Zero GRes) Then $P = (\nu G)\mathbf{0}$ and $Q = \mathbf{0}$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc GRes) with $E, G \vdash \mathbf{0} : F$ and $G \notin fg(F)$. By Lemma A.13, $E \vdash \mathbf{0} : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash \mathbf{0} : F$. We may assume that the bound name G does not occur in $dom(E)$. Hence, by Lemma A.8, $G \notin fg(F)$, and by Lemma A.4, $E, G \vdash \mathbf{0} : F$. By (Proc GRes), $E \vdash (\nu G)\mathbf{0} : F$, that is, $E \vdash P : F$.

(Struct Zero Repl) Then $P = !\mathbf{0}$ and $Q = \mathbf{0}$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc Repl) with $E \vdash \mathbf{0} : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash \mathbf{0} : F$. By (Proc Repl), $E \vdash !\mathbf{0} : F$, that is, $E \vdash P : F$.

(Struct ϵ) Then $P = \epsilon.P'$ and $Q = P'$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc Action) with $E \vdash \epsilon : Cap[F]$ and $E \vdash P' : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash P' : F$. By Lemma A.14, $E \vdash F$. By (Type Cap), $E \vdash Cap[F]$. By (Exp ϵ), $E \vdash \epsilon : Cap[F]$. By (Proc Action), $E \vdash \epsilon.P' : F$, that is, $E \vdash P : F$.

(Struct .) Then $P = (M.M').P'$ and $Q = M.M'.P'$.

For (1), assume $E \vdash P : F$. This must have been derived from (Proc Action) with $E \vdash P' : F$ and $E \vdash M.M' : Cap[F]$. The latter must have come from (Exp .) with $E \vdash M : Cap[F]$ and $E \vdash M' : Cap[F]$, By (Proc Action) twice, $E \vdash M.(M'.P') : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash Q : F$. This must have been derived from (Proc Action), twice, with $E \vdash M : Cap[F]$, $E \vdash M' : Cap[F]$, and $E \vdash P' : F$. By (Exp .), $E \vdash M.M' : Cap[F]$. By (Proc Action), $E \vdash (M.M').P' : F$, that is, $E \vdash P : F$.

(Struct Go ϵ) Then $P = go\ \epsilon.M[P']$ and $Q = M[P']$.

For (1), assume $E \vdash P : F$. This must have been derived using (Proc Go), with $E \vdash F$, $E \vdash \epsilon : Cap[{}^\frown\mathbf{G}, {}^\circ\{\}, Shh]$, $E \vdash M : G{}^\frown\mathbf{G}[F']$, and $E \vdash P' : F'$. By (Proc Amb), $E \vdash M[P'] : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash Q : F$. This must have been derived using (Proc Amb), with $E \vdash F$, $E \vdash P' : F'$ and $E \vdash M : G{}^\frown\mathbf{G}[F']$. By Lemma A.5, $E \vdash G{}^\frown\mathbf{G}[F']$. This must have been derived using (Type Amb), with $\mathbf{G} \subseteq dom(E)$, and $E \vdash F'$. By (Effect Shh) and Lemma A.1, $E \vdash {}^\frown\mathbf{G}, {}^\circ\{\}, Shh$. By (Type Cap), $E \vdash Cap[{}^\frown\mathbf{G}, {}^\circ\{\}, Shh]$. By (Exp ϵ), $E \vdash \epsilon : Cap[{}^\frown\mathbf{G}, {}^\circ\{\}, Shh]$. By (Proc Go), $E \vdash go\ \epsilon.M[P'] : F$, that is, $E \vdash P : F$.

(Struct Go ϵ .) Then $P = go\,(\epsilon.M).N[P']$ and $Q = go\,M.N[P']$. This case follows by an argument very similar to the case for (Struct Go ϵ). We omit the details.

(Struct Go . ϵ) Then $P = go\ (M.\epsilon).N[P']$ and $Q = go\ M.N[P']$.

For (1), assume $E \vdash P : F$. This must have been derived using (Proc Go), with $E \vdash F$, $E \vdash M.\epsilon : Cap[F'']$, $E \vdash N : G{}^\frown\mathbf{G}[F']$, and $E \vdash P' : F'$, with $F'' = {}^\frown\mathbf{G}, {}^\circ\{\}, Shh$. The judgment $E \vdash M.\epsilon : Cap[F'']$ must have been derived using (Exp .) from $E \vdash M : Cap[F'']$ and $E \vdash \epsilon : Cap[F'']$. By (Proc Go), we can derive $E \vdash go\ M.N[P'] : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash Q : F$. This must have been derived using (Proc Go), with $E \vdash F$, $E \vdash M : Cap[F'']$, $E \vdash N : G{}^\frown\mathbf{G}[F']$, and $E \vdash P' : F'$, with $F'' = {}^\frown\mathbf{G}, {}^\circ\{\}, Shh$. By (Exp ϵ) and (Exp .), we get $E \vdash M.\epsilon : Cap[F'']$. By (Proc Go), we can derive $E \vdash go\ (M.\epsilon).N[P'] : F$, that is, $E \vdash P : F$.

(Struct Go . Assoc) In this case, we have $P = go\ ((M.M').M'').N[P']$ and $Q = go\ (M.(M'.M'')).N[P']$.

For (1), assume $E \vdash P : F$. This must have been derived using (Proc Go), with $E \vdash F$, $E \vdash (M.M').M'' : Cap[F'']$, $E \vdash N : G \frown \mathbf{G}[F']$, and $E \vdash P' : F'$, with $F'' = \frown\mathbf{G}, ^\circ\{\}, Shh$. The judgment $E \vdash (M.M').M'' : Cap[F'']$ must have been derived using (Exp .), twice from $E \vdash M : Cap[F'']$ and $E \vdash M' : Cap[F'']$ and $E \vdash M'' : Cap[F'']$. By (Exp .), we can derive $E \vdash M.(M'.M'') : Cap[F'']$, and then, by (Proc Go), we can derive $E \vdash go\ (M.(M'.M'')).N[P'] : F$, that is, $E \vdash Q : F$.

For (2), assume $E \vdash Q : F$. This must have been derived using (Proc Go), with $E \vdash F$, $E \vdash M.(M'.M'') : Cap[F'']$, $E \vdash N : G \frown \mathbf{G}[F']$, and $E \vdash P' : F'$, with $F'' = \frown\mathbf{G}, ^\circ\{\}, Shh$. The judgment $E \vdash M.(M'.M'') : Cap[F'']$ must have been derived using (Exp .), twice from $E \vdash M : Cap[F'']$ and $E \vdash M' : Cap[F'']$ and $E \vdash M'' : Cap[F'']$. By (Exp .), we can derive $E \vdash (M.M').M'' : Cap[F'']$, and then, by (Proc Go), we can derive $E \vdash go\ ((M.M').M'').N[P'] : F$, that is, $E \vdash P : F$.

■

Proof of Theorem 6.1 *If $E \vdash P : F$ and $P \to Q$ then there are $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : F$.*

Proof. For the sake of conciseness, throughout this proof the fact that $E \vdash P : F$ implies $E \vdash F$ (Lemma A.14) will be used several times, without any further explicit acknowledgement. We proceed by induction on the derivation of $P \to Q$.

(Red In) Then $P = n[in\ m.P' \mid P''] \mid m[P''']$ and $Q = m[n[P' \mid P''] \mid P''']$. Assume $E \vdash P : F$. This must have been derived from (Proc Par), with $E \vdash n[in\ m.P' \mid P''] : F$ and $E \vdash m[P'''] : F$. The former must have been derived from (Proc Amb), with $E \vdash F$, $E \vdash n : G_n \frown \mathbf{G}_n[F_n]$ and $E \vdash in\ m.P' \mid P'' : F_n$, for some G_n, $\mathbf{G}_n$, F_n, while the latter must have been derived from (Proc Amb) with $E \vdash F$, $E \vdash m : G_m \frown \mathbf{G}_m[F_m]$ and $E \vdash P''' : F_m$, for some G_m, $\mathbf{G}_m$, F_m. Moreover, $E \vdash in\ m.P' \mid P'' : F_n$ must come from (Proc Par) with $E \vdash in\ m.P' : F_n$ and $E \vdash P'' : F_n$. Finally, $E \vdash in\ m.P' : F_n$ must come from $E \vdash in\ m : Cap[F_n]$ and $E \vdash P' : F_n$. By (Proc Par), we have $E \vdash P' \mid P'' : F_n$, and by (Proc Amb) we can derive $E \vdash n[P' \mid P''] : F_m$. Then, by (Proc Par), we have $E \vdash n[P' \mid P''] \mid P''' : F_m$. By (Proc Amb) we can derive $E \vdash m[n[P' \mid P''] \mid P'''] : F$, that is, $E \vdash Q : F$.

(Red Out) Then $P = m[n[out\ m.P' \mid P''] \mid P''']$ and $Q = n[P' \mid P''] \mid m[P''']$. Assume $E \vdash P : F$. This must have been derived using (Proc Amb) from $E \vdash F$, $E \vdash m : G_m \frown \mathbf{G}_m[F_m]$ and $E \vdash n[out\, m.P' \mid P''] \mid P''' : F_m$ for some G_m, $\mathbf{G}_m$, F_m, and from (Proc Par) using $E \vdash n[outm.P' \mid P''] : F_m$ and $E \vdash P''' : F_m$. The former must have been derived using (Proc Amb) from $E \vdash F_m$, $E \vdash n : G_n \frown \mathbf{G}_n[F_n]$ and $E \vdash out\ m.P' \mid P'' : F_n$ for some G_n, $\mathbf{G}_n$, F_n, and using (Proc Par) from $E \vdash out\ m.P' : F_n$ and $E \vdash P'' : F_n$. The former must have been derived using (Proc Action) from $E \vdash out\ m : Cap[F_n]$ and $E \vdash P' : F_n$. By (Proc Par), $E \vdash P' \mid P'' : F_n$. By (Proc Amb), $E \vdash n[P' \mid P''] : F$. By (Proc Amb), $E \vdash m[P'''] : F$. By (Proc Par), $E \vdash n[P' \mid P''] \mid m[P'''] : F$, that is, $E \vdash Q : F$.

(Red Open) Then $P = open\ n.P' \mid n[P'']$ and $Q = P' \mid P''$. Assume $E \vdash P : F$. This must have been derived using (Proc Par) from $E \vdash open\ n.P' : F$ and

$E \vdash n[P''] : F$. The former must have been derived using (Proc Action) with $E \vdash \mathit{open}\, n : \mathit{Cap}[F]$ and $E \vdash P' : F$, while the latter must have been derived using (Proc Amb) with $E \vdash F$, $E \vdash n : G' \frown \mathbf{G}'[F']$ and $E \vdash P'' : F'$ for some G', $\mathbf{G}'$, F'. The judgment $E \vdash \mathit{open}\, n : \mathit{Cap}[F]$ must have been derived using (Exp Open) from $E \vdash n : G \frown \mathbf{G}[F]$ for some G, $\mathbf{G}$. By Lemma A.7, $G' \frown \mathbf{G}'[F'] = G \frown \mathbf{G}[F]$, and so, in particular, $F' = F$. Hence, by (Proc Par), $E \vdash P' \mid P'' : F$, that is, $E \vdash Q : F$.

(Red I/O) In this case we have $P = (n_1{:}W_1, \ldots, n_k{:}W_k).P' \mid \langle M_1, \ldots, M_k \rangle$ and $Q = P'\{n_1 \leftarrow M_1, \ldots, n_k \leftarrow M_k\}$. Assume $E \vdash P : F$. This must have been derived from (Proc Par) with $E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).P' : F$ and $E \vdash \langle M_1, \ldots, M_k \rangle : F$. The former can only have been derived from (Proc Input) with $E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P' : F$ and $F = {\frown}\mathbf{G}, {^\circ}\mathbf{H}, W_1 \times \cdots \times W_k$ for some $\mathbf{G}$, $\mathbf{H}$. The latter judgment $E \vdash \langle M_1, \ldots, M_k \rangle : F$ must have been derived from (Proc Output) with $E \vdash M_i : W_i'$ for each $i \in 1..k$, and $F = {\frown}\mathbf{G}, {^\circ}\mathbf{H}, W_1' \times \cdots \times W_k'$. Hence $W_i' = W_i$ for each $i \in 1..k$. By k applications of Lemma A.19, we get $E \vdash P'\{n_1 \leftarrow M_1, \ldots, n_k \leftarrow M_k\} : F$.

(Red Go In) Here $P = \mathit{go}(\mathit{in}\, m.N).n[P_n] \mid m[P_m]$ and $Q = m[\mathit{go}\, N.n[P_n] \mid P_m]$. Assume $E \vdash P : F$. This must have been derived using (Proc Par) from $E \vdash \mathit{go}(\mathit{in} m.N).n[P_n] : F$ and $E \vdash m[P_m] : F$. The former must have been derived using (Proc Go) with $E \vdash F$, $E \vdash \mathit{in}\, m.N : \mathit{Cap}[{\frown}\mathbf{G}_n, {^\circ}\{\}, \mathit{Shh}]$, $E \vdash n : G_n \frown \mathbf{G}_n[F_n]$, and $E \vdash P_n : F_n$ for some G_n, $\mathbf{G}_n$, F_n,, and the latter must have been derived using (Proc Amb) with $E \vdash F$, $E \vdash m : G_m \frown \mathbf{G}_m[F_m]$ and $E \vdash P_m : F_m$ for some G_m, $\mathbf{G}_m$, F_m. Moreover, the judgment $E \vdash \mathit{in}\, m.N : \mathit{Cap}[{\frown}\mathbf{G}_n, {^\circ}\{\}, \mathit{Shh}]$ must have been derived using (Exp .) from $E \vdash \mathit{in}\, m : \mathit{Cap}[{\frown}\mathbf{G}_n, {^\circ}\{\}, \mathit{Shh}]$ and $E \vdash N : \mathit{Cap}[{\frown}\mathbf{G}_n, {^\circ}\{\}, \mathit{Shh}]$. By (Proc Go) and (Proc Par), $E \vdash \mathit{go}\, N.n[P_n] \mid P_m : F_m$. By (Proc Amb), we get $E \vdash m[\mathit{go}\, N.n[P_n] \mid P_m] : F$, that is, $E \vdash Q : F$.

(Red Go Out) Here $P = m[\mathit{go}(\mathit{out}\, m.N).n[P_n] \mid P_m]$ and $Q = \mathit{go}\, N.n[P_n] \mid m[P_m]$. Assume $E \vdash P : F$. This must have been derived using (Proc Amb) from $E \vdash F$, $E \vdash m : G_m \frown \mathbf{G}_m[F_m]$ and $E \vdash \mathit{go}(\mathit{out}\, m.N).n[P_n] \mid P_m : F_m$ for some G_m, $\mathbf{G}_m$, F_m, and from (Proc Par) with $E \vdash \mathit{go}(\mathit{out} m.N).n[P_n] : F_m$ and $E \vdash P_m : F_m$. The former must have been derived using (Proc Go) from $E \vdash F_m$, $E \vdash \mathit{out}\, m.N : \mathit{Cap}[{\frown}\mathbf{G}_n, {^\circ}\{\}, \mathit{Shh}]$, $E \vdash n : G_n \frown \mathbf{G}_n[F_n]$, and $E \vdash P_n : F_n$ for some G_n, $\mathbf{G}_n$, F_n. The judgment $E \vdash \mathit{out}\, m.N : \mathit{Cap}[{\frown}\mathbf{G}_n, {^\circ}\{\}, \mathit{Shh}]$ must have been derived using (Proc .) from $E \vdash \mathit{out}\, m : \mathit{Cap}[{\frown}\mathbf{G}_n, {^\circ}\{\}, \mathit{Shh}]$ and $E \vdash N : \mathit{Cap}[{\frown}\mathbf{G}_n, {^\circ}\{\}, \mathit{Shh}]$. By (Proc Go), $E \vdash \mathit{go}\, N.n[P_n] : F$. By (Proc Amb), $E \vdash m[P_m] : F$. By (Proc Par), $E \vdash \mathit{go}\, N.n[P_n] \mid m[P_m] : F$, that is, $E \vdash Q : F$.

(Red Res) Here $P = (\nu n{:}W)P'$ and $Q = (\nu n{:}W)Q'$ with $P' \to Q'$. Assume $E \vdash P : F$. This must have been derived using (Proc Res) from $E, n{:}G \frown \mathbf{G}'[F'] \vdash P' : F$ with $W = G \frown \mathbf{G}'[F']$. By induction hypothesis, $\exists \mathbf{G}$ such that $\mathbf{G}, E, n{:}G \frown \mathbf{G}'[F'] \vdash Q' : F$. By (Proc Res), $\mathbf{G}, E \vdash (\nu n{:}G \frown \mathbf{G}'[F'])Q' : F$, that is, $\mathbf{G}, E \vdash Q : F$.

(Red GRes) Here $P = (\nu G)P'$ and $Q = (\nu G)Q'$ with $P' \to Q'$. Assume $E \vdash P : F$. This must have been derived using (Proc GRes) from $E, G \vdash P' : F$ with $G \notin \mathit{fg}(F)$. By induction hypothesis, $\exists \mathbf{G}$ such that $\mathbf{G}, E, G \vdash Q' : F$. By (Proc GRes), $\mathbf{G}, E \vdash (\nu G)Q' : F$, that is, $\mathbf{G}, E \vdash Q : F$.

(Red Amb) Here $P = n[P']$ and $Q = n[Q']$ with $P' \to Q'$. Assume $E \vdash P : F$. This must have been derived using (Proc Amb) from $E \vdash F$, $E \vdash n : G \frown \mathbf{G}'[F']$, and $E \vdash P' : F'$. By induction hypothesis, $\exists \mathbf{G}$ such that $\mathbf{G}, E \vdash Q' : F'$. By Lemma A.6, $\mathbf{G}, E \vdash F$ and $\mathbf{G}, E \vdash n : G \frown \mathbf{G}'[F']$. By (Proc Amb), $\mathbf{G}, E \vdash n[Q'] : F$, that is, $\mathbf{G}, E \vdash Q : F$.

(Red Par) Here $P = P' \mid R$ and $Q = Q' \mid R$ with $P' \rightarrow Q'$. Assume $E \vdash P : F$. This must have been derived using (Proc Par) from $E \vdash P' : F$ and $E \vdash R : F$. By induction hypothesis, $\exists \mathbf{G}$ such that $\mathbf{G}, E \vdash Q' : F$. By Lemma A.6, $\mathbf{G}, E \vdash R : F$. By (Proc Par), $\mathbf{G}, E \vdash Q' \mid R : F$, that is, $\mathbf{G}, E \vdash Q : F$.

(Red $\equiv$) Here $P \equiv P'$, $P' \rightarrow Q'$, and $Q' \equiv Q$. Assume $E \vdash P : F$. By Proposition A.1, $\exists \mathbf{G}_1$ such that $\mathbf{G}_1, E \vdash P' : F$. By induction hypothesis, $\exists \mathbf{G}_2$ such that $\mathbf{G}_2, \mathbf{G}_1, E \vdash Q' : F$. By Proposition A.1, $\exists \mathbf{G}_3$ such that $\mathbf{G}_3, \mathbf{G}_2, \mathbf{G}_1, E \vdash Q : F$.

■

APPENDIX B

Proof of Effect Safety

In this appendix we prove the effect safety property stated in Section 7.

Proof of Proposition 7.1 *Suppose that* $E \vdash P : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$.

(1) *If* $P \downarrow \mathit{in}\, n$ *then* $E \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$ *for some type* $G\,{}^{\frown}\mathbf{G}'[F]$ *with* $G \in \mathbf{G}$.

(2) *If* $P \downarrow \mathit{out}\, n$ *then* $E \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$ *for some type* $G\,{}^{\frown}\mathbf{G}'[F]$ *with* $G \in \mathbf{G}$.

(3) *If* $P \downarrow \mathit{open}\, n$ *then* $E \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$ *for some type* $G\,{}^{\frown}\mathbf{G}'[F]$ *with* $G \in \mathbf{H}$.

Proof. We prove part (1) in detail; the other parts follow by similar arguments. We proceed by induction on the derivation of $P \downarrow \mathit{in}\, n$.

(Ex Cap) We have $P \downarrow \mathit{in}\, n$ derived from $P \equiv \mathit{in}\, n.Q$. By Proposition A.1, $E \vdash P : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$ and $P \equiv \mathit{in}\, n.Q$ imply there are groups $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash \mathit{in}\, n.Q : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$. This must have been derived using (Proc Action) from $G_1, \ldots, G_k, E \vdash \mathit{in}\, n : \mathit{Cap}[{}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T]$, which itself must have been derived using (Exp In) from $G_1, \ldots, G_k, E \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$ for some type $G\,{}^{\frown}\mathbf{G}'[F]$ with $G \in \mathbf{G}$. The latter judgment must have been derived using (Exp n), and therefore $E = E', n{:}G\,{}^{\frown}\mathbf{G}'[F], E''$. By Lemma A.1, $E \vdash P : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$ implies $E \vdash \diamond$, and therefore $E \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$, by (Exp n).

(Ex Par 1) We have $P \mid Q \downarrow \mathit{in}\, n$ derived from $P \downarrow \mathit{in}\, n$. The judgment $E \vdash P \mid Q : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$ must have been derived using (Proc Par) from $E \vdash P : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$. By induction hypothesis, this and $P \downarrow \mathit{in}\, n$ imply the required result.

(Ex Par 2) We have $P \mid Q \downarrow \mathit{in}\, n$ derived from $Q \downarrow \mathit{in}\, n$. The judgment $E \vdash P \mid Q : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$ must have been derived using (Proc Par) from $E \vdash Q : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$. By induction hypothesis, this and $Q \downarrow \mathit{in}\, n$ imply the required result.

(Ex Res) We have $(\nu m{:}W)P \downarrow \mathit{in}\, n$ derived from $P \downarrow \mathit{in}\, n$ and $m \notin \mathit{fn}(\mathit{in}\, n)$. The judgment $E \vdash (\nu m{:}W)P : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$ must have been derived using (Proc Res) from $E, m{:}W \vdash P : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$. By induction hypothesis, this and $P \downarrow \mathit{in}\, n$ imply that $E, m{:}W \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$ for some type $G\,{}^{\frown}\mathbf{G}'[F]$ with $G \in \mathbf{G}$. By Lemma A.12, $m \neq n$ and $E, m{:}W \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$ imply $E \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$.

(Ex ResG) We have $(\nu H)P \downarrow \mathit{in}\, n$ derived from $P \downarrow \mathit{in}\, n$. The judgment $E \vdash (\nu H)P : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$ must have been derived using (Proc GRes) from $E, H \vdash P : {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$ with $H \notin \mathit{fg}({}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T)$. By induction hypothesis, the latter and $P \downarrow \mathit{in}\, n$ imply that $E, H \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$ for some type $G\,{}^{\frown}\mathbf{G}'[F]$ with $G \in \mathbf{G}$. By Lemma A.13, $H \notin \mathit{fg}({}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T)$ and $E, H \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$ imply $E \vdash n : G\,{}^{\frown}\mathbf{G}'[F]$.

■

REFERENCES

1. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.
2. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
3. R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings COORDINATION 97*, volume 1282 of *Lecture Notes in Computer Science*. Springer, 1997.
4. R. M. Amadio and S. Prasad. Localities and failures. In *Proceedings FST&TCS'94*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 1994.
5. T. Amtoft, A. J. Kfoury, and S. M. Pericas-Geertsen. What are polymorphically-typed ambients? In *Proceedings ESOP'01*, Lecture Notes in Computer Science, pages 206–220. Springer, 2001.
6. C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Control flow analysis for the π-calculus. In *Proceedings Concur'98*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1998.
7. G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.
8. M. Bugliesi and G. Castagna. Secure safe ambients. In *Proceedings POPL'01*, pages 222–235. ACM, January 2001.
9. L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In *Proceedings ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 230–239. Springer, 1999.
10. L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *Proceedings IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2000.
11. L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In C. Palamidessi, editor, *CONCUR 2000—Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2000.
12. L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings POPL'99*, pages 79–92. ACM, January 1999.
13. L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000.
14. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
15. S. Dal Zilio and A. D. Gordon. Region analysis and a π-calculus with groups. In *Mathematical Foundations of Computer Science 2000*, volume 1893 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2000. Accepted for publication in the *Journal of Functional Programming*.
16. R. De Nicola, G. Ferrari, and R. Pugliese. Types as specifications of access policies. In *Secure Internet Programming 1999*, volume 1603 of *Lecture Notes in Computer Science*, pages 117–146. Springer, 1999.
17. P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Advances in Computing Science (ASIAN'00)*, volume 1961 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2000.
18. M. Dezani-Ciancaglini and I. Salvo. Security types for mobile safe ambients. In *Advances in Computing Science (ASIAN'00)*, volume 1961 of *Lecture Notes in Computer Science*, pages 215–236. Springer, 2000.
19. J. Feret. Abstract interpretation-based static analysis of mobile ambients. In *Static Analysis (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 412–430. Springer, 2001.
20. R.R. Hansen, J.G. Jensen, F. Nielson, and H. Riis Nielson. Abstract interpretation of mobile ambients. In *Static Analysis (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 1999.
21. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS, pages 133–147. Springer-Verlag, 1991.
22. D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

23. F. Levi and C. Bodei. Security analysis for mobile ambients. In *Workshop on Issues in the Theory of Security (WITS'00)*, pages 18–23, 2000.

24. F. Levi and S. Maffeis. An abstract interpretation framework for analysing mobile ambients. In *Static Analysis (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 395–411. Springer, 2001.

25. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings POPL'00*, pages 352–364. ACM, 2000.

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

27. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999.

28. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.

29. F. Nielson, H. Riis Nielson, R.R. Hansen, and J.G. Jensen. Validating firewalls in mobile ambients. In *Concurrency Theory (Concur'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 463–477. Springer, 1999.

30. F. Nielson, H. Riis Nielson, and M. Sagiv. Abstract interpretation of mobile ambients. In *Programming Languages and Systems (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2000.

31. H. Riis Nielson and F. Nielson. Shape analysis for mobile ambients. In *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 135–148, 2000.

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

33. P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 1998.

34. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

35. J.E. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/The MIT Press, 1996.

36. P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *Proceedings of Foundations of Software Science and Computation Structures FOSSACS'00*, volume 1784 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2000.