Types for Mobile Ambients

Luca Cardelli
Andrew D. Gordon
Microsoft Research

Abstract

Java has demonstrated the utility of type systems for *mobile code*, and in particular their use and implications for security. Security properties rest on the fact that a well-typed Java program (or the corresponding verified bytecode) cannot cause certain kinds of damage.

In this paper we provide a type system for *mobile computation*, that is, for computation that is continuously active before and after movement. We show that a well-typed mobile computation cannot cause certain kinds of run-time fault: it cannot cause the exchange of values of the wrong kind, anywhere in a mobile system.

1 Introduction

In previous work [4] we introduced the (untyped, monadic) *ambient calculus*, a process calculus for mobile computation and mobile devices. That calculus is able to express, via encodings, standard computational constructions such as channel-based communication, functions, and agents.

The type system presented in this paper is able to provide typings for those encodings, recovering familiar type systems for processes and functions. In addition, we obtain a type system for mobile agents and other mobile computations. The type system is obtained by decorating the untyped calculus with type information.

An *ambient*, in our sense, is a confined place where processes run. Each ambient has a name, and may contain multiple processes and subambients. A process can cause its surrounding ambient to move in or out of other ambients, transporting all the subambients and active processes with it. A process may also *open* an ambient, that is, it can dissolve an ambient boundary while preserving its contents. Finally, processes within the same ambient may exchange messages.

Our type system tracks the typing of messages exchanged within an ambient. For example, the following system consists of two ambients, named a and b:

$$a[(x{:}Int).P \mid open\ b] \mid b[in\ a.\ \langle 3\rangle]$$

The ambient named a contains a process $(x{:}Int).P$ that is ready to read an integer message into a variable x and proceed with P, and a process *open b* that is ready to open (dissolve the boundary) of an ambient b found within a. The ambient named b contains a process *in a*. $\langle 3\rangle$ that moves the ambient b inside a (by executing *in a*) and then outputs the message 3. The ambient b is opened after moving into a, so the output comes into direct contact with the reading process within a. The result is the binding of an integer message to an integer variable, yielding the state:

$$a[P\{x{\leftarrow}3\}]$$

The challenge of the type system is to verify that this exchange of messages is well-typed. Note that in the original system the input and the output were contained in separate locations.

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 [1] prevents a channel from being defined at more than one location. Sewell's system [12] tracks whether communications are local or non-local, so as to allow efficient implementation of local communication. In Riely and Hennessy's calculus [11], 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 [5] that tracks the access rights an agent enjoys at different localities; type-checking ensures that an agent complies with its access rights.

2 The Polyadic Ambient Calculus

We begin by reviewing and slightly extending the ambient calculus of [4]. 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. In addition, 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 [4] 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, **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 [2, 6], except that the channel has no name: the surrounding ambient provides the context where the communication happens. The process $\langle M_1, ..., M_k\rangle$ represents the output of a tuple of values, with no continuation. The process $(n_1{:}W_1, ..., n_k{:}W_k).P$ represents the input of a tuple of values, with continuation P.

Communication is used to exchange both names and capabilities, which share the same syntactic class M of expressions. One of the main tasks 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.

Polyadic Ambient Calculus

$P,Q ::=$	processes
$(\nu n{:}W)P$	restriction
$\mathbf{0}$	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	action
$(n_1{:}W_1, ..., n_k{:}W_k).P$	input
$\langle M_1, ..., M_k\rangle$	async output
$M ::=$	expressions
n	name
$in\ M$	can enter into M
$out\ M$	can exit out of M
$open\ M$	can open M
ε	null path
$M.M'$	composite path

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	$(M.P) \mid Q$	
$(n_1{:}W_1, ..., n_k{:}W_k).P \mid Q$	is read	$((n_1{:}W_1, ..., n_k{:}W_k).P) \mid Q$	
$(\nu n_1{:}W_1, ..., n_k{:}W_k)P$	$\triangleq$	$(\nu n_1{:}W_1)...(\nu n_k{:}W_k)P$	
$n[]$	$\triangleq$	$n[\mathbf{0}]$	
M	$\triangleq$	$M.\mathbf{0}$	(where appropriate)

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 and will be explained in the next section.

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

Terms are also identified up to the consistent renaming of bound variables, in the restriction and input constructs. We write $P\{n \leftarrow M\}$ for the substitution of M for each free occurrence of the name n in the process P. Similarly for $M\{n \leftarrow M'\}$.

Free names

$fn((\nu n{:}W)P) \triangleq fn(P) - \{n\}$
$fn(\mathbf{0}) \triangleq \emptyset$
$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$
$fn(!P) \triangleq fn(P)$
$fn(M[P]) \triangleq fn(M) \cup fn(P)$
$fn(M.P) \triangleq fn(M) \cup fn(P)$
$fn((n_1{:}W_1, ..., n_k{:}W_k).P) \triangleq fn(P) - \{n_1, ..., n_k\}$
$fn(\langle M_1, ..., M_k\rangle) \triangleq fn(M_1) \cup ... \cup fn(M_k)$
$fn(n) \triangleq \{n\}$
$fn(in\ M) \triangleq fn(M)$
$fn(out\ M) \triangleq fn(M)$
$fn(open\ M) \triangleq fn(M)$
$fn(\varepsilon) \triangleq \emptyset$
$fn(M.M') \triangleq fn(M) \cup fn(M')$

Structural Congruence

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

$P \equiv Q \;\Rightarrow\; (n_1{:}T_1, ..., n_k{:}T_k).P \equiv (n_1{:}T_1, ..., n_k{:}T_k).Q$	(Struct Input)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu n{:}T)(\nu m{:}U)P \equiv (\nu m{:}U)(\nu n{:}T)P \quad \text{if } n \neq m$	(Struct Res Res)
$(\nu n{:}T)(P \mid Q) \equiv P \mid (\nu n{:}T)Q \quad \text{if } n \notin fn(P)$	(Struct Res Par)
$(\nu n{:}T)m[P] \equiv m[(\nu n{:}T)P] \quad \text{if } n \neq m$	(Struct Res Amb)
$P \mid \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n{:}Amb[T])\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Res)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Repl)
$\varepsilon.P \equiv P$	(Struct ε)
$(M.M').P \equiv M.M'.P$	(Struct .)

Reduction

$n[in\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$	(Red Open)
$(n_1{:}W_1, ..., n_k{:}W_k).P \mid \langle M_1, ..., M_k\rangle \longrightarrow P\{n_1{\leftarrow}M_1, ..., n_k{\leftarrow}M_k\}$	(Red Comm)
$P \longrightarrow Q \;\Rightarrow\; (\nu n{:}W)P \longrightarrow (\nu n{:}W)Q$	(Red Res)
$P \longrightarrow Q \;\Rightarrow\; n[P] \longrightarrow n[Q]$	(Red Amb)
$P \longrightarrow Q \;\Rightarrow\; P \mid R \longrightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \longrightarrow Q, Q \equiv Q' \;\Rightarrow\; P' \longrightarrow Q'$	(Red ≡)

3 Exchange Types

An ambient is a place where other ambients can enter and exit, and where processes can exchange messages. The first aspect, mobility, is regulated by run-time capabilities and will not be restricted by our type system. The second aspect, communication, is what we now concentrate on.

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 our 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 $Amb[T]$, the type of names of ambients 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 $W_1 \times ... \times W_k$, the exchange of a tuple of messages with elements of the respective message types. For k=0, the empty tuple type is called **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 types
$Amb[T]$	ambient name allowing T exchange
$Cap[T]$	capability unleashing T exchange
$T ::=$	exchange types
Shh	no exchange
$W_1 \times ... \times W_k$	tuple exchange

For example:

- A quiet ambient: $Amb[Shh]$
- A harmless capability: $Cap[Shh]$
- A synchronization ambient: $Amb[\mathbf{1}]$
- An ambient that allows the exchange of harmless capabilities: $Amb[Cap[Shh]]$
- A capability that may unleash the exchange of names of quiet ambients: $Cap[Amb[Shh]]$

3.2 Intuitions

Before presenting the formal type rules, we discuss the intuitions that lead to them.

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 will have a rule stating that:

$$M : W \;\Rightarrow\; \langle M\rangle : W$$

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

$$P : W \;\Rightarrow\; (x{:}W).P : W$$

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

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

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

$$P : T, Q : T \;\Rightarrow\; P \mid Q : T$$
$$P : T \;\Rightarrow\; !P : T$$

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.

Typing of Ambients

An ambient $n[P]$ is a process that exchanges nothing at the current level, so, like **0**, it can have any exchange type, and can be placed in parallel with any process.

$n[P] : T \quad$ 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 a type $Amb[T]$, meaning that only T exchanges are allowed in any ambient of that name. Ambients of different names may permit internal exchanges of different types.

$n : Amb[T], P : T \;\Rightarrow$
$n[P]$ 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.

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 *open n* may unleash those T exchanges. We then say that *open n* has a capability type $Cap[T]$, meaning that it may unleash T exchanges when used:

$n : Amb[T] \;\Rightarrow\; open\ n : Cap[T]$

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

$M : Cap[T], P : T \;\Rightarrow\; M.P : T$

The capability types $Cap[T]$ do not keep track of any information concerning *in* and *out* capabilities; only the effect of *open* is tracked.

3.3 Typing Rules

We base our type system on three judgments. The main judgment tracks the exchange type of a process, that is the type of the I/O operations of the process, and of the I/O operations that the process may unleash by opening other ambients.

Judgments

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

Based on the discussion in the previous section, we can formalize the type system as described in the following table. Convention: a list of assumptions $E \vdash J_1 \ldots E \vdash J_k$ for k=0 means $E \vdash \Diamond$.

Rules

(Env ∅) $$\frac{}{\varnothing \vdash \Diamond}$$

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

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

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

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

(Exp In) $$\frac{E \vdash M : Amb[S]}{E \vdash in\ M : Cap[T]}$$

(Exp Out) $$\frac{E \vdash M : Amb[S]}{E \vdash out\ M : Cap[T]}$$

(Exp Open) $$\frac{E \vdash M : Amb[T]}{E \vdash open\ M : Cap[T]}$$

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

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

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

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

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

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

• Example: A process that outputs names of quiet ambients:

$\varnothing \vdash !(\nu n{:}Amb[Shh])\langle n \rangle : Amb[Shh]$

• Example: A capability that may unleash S-exchanges. Note that the *in n* action contributes nothing to the type of the path; only the *open m* action does:

$\varnothing, n{:}Amb[T], m{:}Amb[S] \vdash in\ n.\ open\ m : Cap[S]$

The correctness of the type system is expressed by the following proposition (the proof is in Appendix 7):

3-1 Proposition (Subject Reduction)
If $E \vdash P : U$ and $P \longrightarrow Q$ then $E \vdash Q : U$.
□

Certain "run-time error" expressions are allowed in the syntax but are nonsensical because they confuse names with capabilities. Examples are *in* $n[P]$, $(\nu n{:}Amb[T])n.P$, and $\langle in\ (in\ n)\rangle$. Such expressions are not initially typeable, and they cannot be produced by well-typed processes because Proposition 3-1 says that the evolution of well-typed processes leads only to well-typed processes.

4 Applications

4.1 Channel Types

We now begin to explore the expressiveness of our type system. The first test case is whether we can represent typed communication channels, that is, whether we can find a typed encoding of the π-calculus [8].

The basic idea for the encoding of channels is to use an ambient as a buffer where input and output processes can exchange messages. An output operation generates an output packet that enters the buffer and (after being opened) deposits an output. An input operation generates an input packet that similarly enters the buffer, reads an input, and creates a return packet that exits the buffer and continues with the rest of the process. Each name n of the π-calculus becomes a pair of names in the ambient calculus: the name n of the buffer and the name n^p of the packets. Therefore, communication of a π-calculus name becomes the communication of a pair of ambient calculus names. A π-calculus channel type $Ch[W]$ for names of type W is translated as $Amb[W{\times}W]$.

Encoding of the Typed Polyadic Asynchronous π-calculus

$$\langle\!\langle E \vdash P \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle \vdash \langle\!\langle P \rangle\!\rangle : Shh$$

$$\langle\!\langle \varnothing, n_1{:}W_1, ..., n_k{:}W_k \rangle\!\rangle \triangleq \varnothing, n_1{:}\langle\!\langle W_1 \rangle\!\rangle, n_1{}^p{:}\langle\!\langle W_1 \rangle\!\rangle, ..., n_k{:}\langle\!\langle W_k \rangle\!\rangle, n_k{}^p{:}\langle\!\langle W_k \rangle\!\rangle$$

$$\langle\!\langle Ch[W_1, ..., W_k] \rangle\!\rangle \triangleq Amb[\langle\!\langle W_1 \rangle\!\rangle{\times}\langle\!\langle W_1 \rangle\!\rangle{\times}...{\times}\langle\!\langle W_k \rangle\!\rangle{\times}\langle\!\langle W_k \rangle\!\rangle]$$

$$\langle\!\langle (\nu^\pi n{:}Ch[W_1, ..., W_k])P \rangle\!\rangle \triangleq (\nu n,n^p{:}\langle\!\langle Ch[W_1, ..., W_k] \rangle\!\rangle)\ (n[!open\ n^p] \mid \langle\!\langle P \rangle\!\rangle)$$

$$\langle\!\langle n(n_1{:}W_1, ..., n_k{:}W_k).P \rangle\!\rangle \triangleq (\nu p{:}Amb[Shh])\ (open\ p \mid n^p[in\ n.\ (n_1,n^p{}_1{:}\langle\!\langle W_1 \rangle\!\rangle, ..., n_k,n^p{}_k{:}\langle\!\langle W_k \rangle\!\rangle).\ p[out\ n.\ \langle\!\langle P \rangle\!\rangle]])$$

$$\langle\!\langle n\langle n_1, ..., n_k \rangle \rangle\!\rangle \triangleq n^p[in\ n.\ \langle n_1, n^p{}_1, ..., n_k, n^p{}_k \rangle]$$

$$\langle\!\langle P \mid Q \rangle\!\rangle \triangleq \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle$$

$$\langle\!\langle !P \rangle\!\rangle \triangleq !\langle\!\langle P \rangle\!\rangle$$

The translation induces the following derived typing rules, which correspond to a fragment of Pierce and Sangiorgi's system [10] consisting only of bidirectional channels, with no subtyping. Each π-calculus process is given the type Shh, since no communication happens at the level of processes. Instead, communication happens within buffers, so each buffer receives the type of the corresponding π-calculus channel. Input and output packets receive the same type as the buffers where they are opened.

$$\langle\!\langle E,n{:}Ch[W_1, ..., W_k] \vdash P \rangle\!\rangle \Rightarrow \langle\!\langle E \vdash (\nu^\pi n{:}Ch[W_1, ..., W_k])P \rangle\!\rangle$$

$$\langle\!\langle E \vdash n : Ch[W_1, ..., W_k] \rangle\!\rangle, \langle\!\langle E \vdash n_1 : W_1 \rangle\!\rangle, ..., \langle\!\langle E \vdash n_k : W_k \rangle\!\rangle \Rightarrow \langle\!\langle E \vdash n\langle n_1, ..., n_k \rangle \rangle\!\rangle$$

$$\langle\!\langle E \vdash n : Ch[W_1, ..., W_k] \rangle\!\rangle, \langle\!\langle E, n_1{:}W_1, ..., n_k{:}W_k \vdash P \rangle\!\rangle \Rightarrow \langle\!\langle E \vdash n(n_1{:}W_1, ..., n_k{:}W_k).P \rangle\!\rangle$$

$$\langle\!\langle E \vdash P \rangle\!\rangle, \langle\!\langle E \vdash Q \rangle\!\rangle \Rightarrow \langle\!\langle E \vdash P \mid Q \rangle\!\rangle$$

$$\langle\!\langle E \vdash P \rangle\!\rangle \Rightarrow \langle\!\langle E \vdash !P \rangle\!\rangle$$

Georges Gonthier has devised two other encodings of the π-calculus as ambients. The first encoding uses a single name n for both the buffer and the associated packets, instead of pairs of names n, n^p. The packets are temporarily hidden inside another layer of ambients, so that there is no confusion between packets and buffers. For the π-calculus this techniques leads to a nicer encoding, where a channel type maps simply to an ambient type. Still, the technique of passing packet names along with associated ambient names is often useful, as we show in later examples.

Gonthier's Encoding

$$\langle\!\langle Ch[W_1, ..., W_k] \rangle\!\rangle \triangleq Amb[\langle\!\langle W_1 \rangle\!\rangle{\times}...{\times}\langle\!\langle W_k \rangle\!\rangle]$$

$$\langle\!\langle (\nu^\pi n{:}Ch[W_1, ..., W_k])P \rangle\!\rangle \triangleq (\nu n{:}\langle\!\langle Ch[W_1, ..., W_k] \rangle\!\rangle)\ n[!open\ n] \mid \langle\!\langle P \rangle\!\rangle$$

$$\langle\!\langle n(n_1{:}W_1, ..., n_k{:}W_k).P \rangle\!\rangle \triangleq (\nu p{:}Amb[Shh])\ (open\ p \mid (\nu k{:}\langle\!\langle Ch[W_1, ..., W_k] \rangle\!\rangle)\ k[in\ n.\ n[out\ k.\ open\ k.\ (n_1{:}\langle\!\langle W_1 \rangle\!\rangle, ..., n_k{:}\langle\!\langle W_k \rangle\!\rangle).\ p[out\ n.\ \langle\!\langle P \rangle\!\rangle]]])$$

$$\langle\!\langle n\langle n_{1W_1}, ..., n_{kW_k} \rangle \rangle\!\rangle \triangleq (\nu k{:}\langle\!\langle Ch[W_1, ..., W_k] \rangle\!\rangle)\ k[in\ n.\ n[out\ k.\ open\ k.\ \langle n_1, ..., n_k \rangle]]$$

$$\langle\!\langle P \mid Q \rangle\!\rangle \triangleq \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle$$

$$\langle\!\langle !P \rangle\!\rangle \triangleq !\langle\!\langle P \rangle\!\rangle$$

(We use a subscript type to indicate the type of a term that, while not available in the term itself, is available in its type derivation.)

Gonthier's second encoding also uses single names for buffers and packets. In addition, the encoding does not rely on buffers being generated at the place of ν: buffers are generated whenever (and wherever!) needed by I/O operations. For the π-calculus this makes little difference, but if we imagine using channels freely within the ambient calculus, then it is important not to rely on a fixed location for the buffer: we may want I/O operations on a channel to interact whenever they occur within the same ambient. The potential problem with this idea is that, since there are multiple buffers, all the output packets may go in one buffer, and all the input packets may go in a different buffer. To solve this problems, the buffers are designed to be self-coalescing. This technique is useful in general, when buffers need to be generated in a decentralized fashion.

Gonthier's Coalescing Encoding

$$\langle\!\langle Ch[W_1, ..., W_k] \rangle\!\rangle \triangleq Amb[\langle\!\langle W_1 \rangle\!\rangle{\times}...{\times}\langle\!\langle W_k \rangle\!\rangle]$$

$$\langle\!\langle (\nu^\pi n{:}Ch[W_1, ..., W_k])P \rangle\!\rangle \triangleq (\nu n{:}\langle\!\langle Ch[W_1, ..., W_k] \rangle\!\rangle)\ \langle\!\langle P \rangle\!\rangle$$

$$\langle\!\langle n(n_1{:}W_1, ..., n_k{:}W_k).P \rangle\!\rangle \triangleq (\nu p{:}Amb[Shh])\ (open\ p.p[] \mid n[!open\ n \mid in\ n \mid (n_1{:}\langle\!\langle W_1 \rangle\!\rangle, ..., n_k{:}\langle\!\langle W_k \rangle\!\rangle).\ p[!out\ n \mid open\ p.\langle\!\langle P \rangle\!\rangle]])$$

$$\langle\!\langle n\langle n_1, ..., n_k \rangle \rangle\!\rangle \triangleq n[!open\ n \mid in\ n \mid \langle n_1, ..., n_k \rangle]$$

$$\langle\!\langle P \mid Q \rangle\!\rangle \triangleq \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle$$

$$\langle\!\langle !P \rangle\!\rangle \triangleq !\langle\!\langle P \rangle\!\rangle$$

4.2 Parent-Child Communication

It is often useful for an ambient to communicate with its parent or its children, as when an agent enters a server and wants to exchange information with it. We now describe such a derived communication mechanism, and how to type it.

Parent-Child I/O

$^{\triangledown}n\langle M\rangle$	parent outputs to child n
$^{\vartriangle}n(x{:}W).P$	child n inputs from parent
$^{\vartriangle}n\langle M\rangle$	child n outputs to parent
$^{\triangledown}n(x{:}W).P$	parent inputs from child n

We could adopt the following reduction rules as primitive:

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

Instead of taking these operators as primitive, it is possible to approximate parent-child I/O with normal ambient I/O. The encoding given below, however, fails to provide the same atomicity guarantees as the reductions above. When using this encoding, parent-child I/O operations are partially sensitive to disruptions of the protocol due to sudden movement of the child. To avoid this problem, the child has to implement its own synchronization.

The encoding of parent-to-child messaging is quite simple, using the child ambient as the communication buffer. Messages from the parent down to a child n^{ch} use packets labeled n^{dn}.

$$^{\triangledown}n\langle M\rangle \;\triangleq\; n^{dn}[in\ n^{ch}.\ \langle M\rangle]$$
$$^{\vartriangle}n(x{:}W).P \;\triangleq\; open\ n^{dn}.\ (x{:}W).\ P$$

This messaging is not sensitive to sudden movement of the child: messages from parent to child may get blocked but do not get lost.

The encoding of child-to-parent messaging, instead, is more problematic. There is a choice of where to put the communication buffer: in the child or in the parent. If the buffer is in the child, the parent has to send a process to fetch the message; such a process may get lost on the way back if the child has moved. If the buffer is in the parent, the child has to send a process to deposit the message; such a process may get lost if the child moves before the (asynchronous) process can get out.

In both cases, though, the child can wait for a confirmation from the parent that the message has reached the parent; this can be done with parent-to-child communication, which is reliable. After the confirmation, the child is free to move.

We describe the case where the buffer is kept in the parent. This arrangement seems more interesting because, with a simple modification, it can be extended to anonymous communication between arbitrary children and a parent.

Each communication from a child n^{ch} to a parent happens within a mailbox n^{box} within the parent; the mailboxes are self-coalescing. Messages from a child n^{ch} up to the parent use packets labeled n^{up} that are sent out of the child and then into n^{box}.

$$^{\vartriangle}n\langle M\rangle.P \;\triangleq\; n^{up}[out\ n^{ch}.\ in\ n^{box}.\ \langle M\rangle]$$
$$\begin{aligned}&^{\triangledown}n(x{:}W).P_{W'} \;\triangleq\\ &\quad (\nu p{:}Amb[W'])\ (open\ p.\ p[\,] \mid\\ &\qquad n^{box}[open\ n^{up}.\ (x{:}W).\ p[out\ n^{box}.\ open\ p.\ P] \mid\\ &\qquad\quad !open\ n^{box} \mid in\ n^{box}])\end{aligned}$$

(The idioms *open p. p*[] and *p*[... *open p. P*] are used to delay the activation of *P* until *P* reaches the proper position.)

The type of names of child ambients that admit parent-child I/O may be denoted by $Amb^{\vartriangle\triangledown}[W]$. This notation can be translated to the ambient calculus by mapping each environment name $n : Amb^{\vartriangle\triangledown}[W]$ to four environment names n^{ch}, n^{up}, n^{dn}, n^{box} : $Amb[W]$, and by mapping each restriction $(\nu n{:}Amb^{\vartriangle\triangledown}[W])\ P$ to the restrictions $(\nu n^{ch}{:}Amb[W])\ (\nu n^{up}{:}Amb[W])\ (\nu n^{dn}{:}Amb[W])\ (\nu n^{box}{:}Amb[W])\ P$.

The derived type rules are as follows.

$$(n : Amb^{\vartriangle\triangledown}[W] \Rightarrow P : T) \;\;\Rightarrow\;\; (\nu n : Amb^{\vartriangle\triangledown}[W]).P : T$$
$$M : W,\ n : Amb^{\vartriangle\triangledown}[W] \;\;\Rightarrow\;\; ^{\triangledown}n\langle M\rangle : U \quad (\text{any } U)$$
$$M : W,\ n : Amb^{\vartriangle\triangledown}[W] \;\;\Rightarrow\;\; ^{\vartriangle}n\langle M\rangle : U \quad (\text{any } U)$$
$$n : Amb^{\vartriangle\triangledown}[W],\ (x : W \Rightarrow P : W) \;\;\Rightarrow\;\; ^{\vartriangle}n(x{:}W).P : W$$
$$n : Amb^{\vartriangle\triangledown}[W],\ (x : W \Rightarrow P : W') \;\;\Rightarrow\;\; ^{\triangledown}n(x{:}W).P : W'$$

4.3 Function Types

By using a typed encoding of channels in the ambient calculus, we can provide typed encodings of λ-calculi simply by using the known encodings of λ-calculi into the π-calculus [9]. For example:

Encoding of the Call-by-Value λ-calculus into the π-calculus

$$\langle\!\langle x\rangle\!\rangle_k \;\triangleq\; k\langle x\rangle$$
$$\langle\!\langle \lambda x.b\rangle\!\rangle_k \;\triangleq\; (\nu^{\pi}n)\ (k\langle n\rangle \mid !n(x, k').\ \langle\!\langle b\rangle\!\rangle_{k'})$$
$$\langle\!\langle b(a)\rangle\!\rangle_k \;\triangleq\; (\nu^{\pi}k', k'')\ (\langle\!\langle b\rangle\!\rangle_{k'} \mid k'(x).\ (\langle\!\langle a\rangle\!\rangle_{k''} \mid k''(y).\ x\langle y, k\rangle))$$

Encoding of the Typed Call-by-Value λ-calculus into the Ambient Calculus

$$\langle\!\langle E \vdash b{:}T\rangle\!\rangle \;\triangleq\; \langle\!\langle E\rangle\!\rangle \vdash (\nu^{\pi}k{:}Ch[\langle\!\langle T\rangle\!\rangle])\ \langle\!\langle b\rangle\!\rangle_k : Shh$$
$$\langle\!\langle \varnothing, x_1{:}A_1, ..., x_l{:}A_l\rangle\!\rangle \;\triangleq\; \varnothing, x_1{:}\langle\!\langle A_1\rangle\!\rangle, ..., x_l{:}\langle\!\langle A_l\rangle\!\rangle$$
$$\langle\!\langle A{\to}B\rangle\!\rangle \;\triangleq\; Ch[\langle\!\langle A\rangle\!\rangle, Ch[\langle\!\langle B\rangle\!\rangle]]$$
$$\langle\!\langle x\rangle\!\rangle_k \;\triangleq\; k\langle x\rangle$$
$$\begin{aligned}&\langle\!\langle \lambda x{:}A.b_{A\to B}\rangle\!\rangle_k \;\triangleq\\ &\quad (\nu^{\pi}n{:}\langle\!\langle A{\to}B\rangle\!\rangle)\ (k\langle n\rangle \mid !n(x{:}\langle\!\langle A\rangle\!\rangle, k'{:}Ch[\langle\!\langle B\rangle\!\rangle]).\ \langle\!\langle b_B\rangle\!\rangle_{k'})\end{aligned}$$
$$\begin{aligned}&\langle\!\langle b_{A\to B}(a_A)\rangle\!\rangle_k \;\triangleq\\ &\quad (\nu^{\pi}k'{:}Ch[\langle\!\langle A{\to}B\rangle\!\rangle], k''{:}Ch[\langle\!\langle A\rangle\!\rangle])\\ &\qquad (\langle\!\langle b\rangle\!\rangle_{k'} \mid k'(x{:}\langle\!\langle A{\to}B\rangle\!\rangle).\ (\langle\!\langle a\rangle\!\rangle_{k''} \mid k''(y{:}\langle\!\langle A\rangle\!\rangle).\ x\langle y, k\rangle))\end{aligned}$$

Therefore, as in the π-calculus, a function is represented by a channel that communicates an argument and a channel for the result. The derived types reflect this structure.

4.4 Records

We define operations for handling records of mutable cells; these will be useful in the next example.

A record r containing cells c_i has the general structure $r[\ ...\ |\ c_i^{buf}[\langle M_i\rangle]\ |\ !open\ c_i^{ip}]\ |\ ...\ ]$, where r is the cell container, c_i^{buf} are the value containers for each cell, c_i^{ip} are input packets for reading and writing cell contents, and M_i are the cell contents. The operations consist of creating an empty record named r (*record r*), adding a cell named c with initial contents M to a record r (*add r c M*), reading the contents of cell c of record r and binding it to a variable x in a scope P (*get r c* $(x{:}W).\ P$), and setting the contents of a cell c of record r to a value M and continuing with P (*set r c* $\langle M\rangle.\ P$).

$$
\begin{aligned}
&\langle\!\langle record\ r\rangle\!\rangle \ \triangleq\ r[\,]\\
&\langle\!\langle add\ r\ c\ M\rangle\!\rangle \ \triangleq\ c^{buf}[!open\ c^{ip}\ |\ in\ r.\ \langle M\rangle]\\
&\langle\!\langle get\ r\ c\ (x{:}W).\ P_S\rangle\!\rangle \ \triangleq\\
&\quad (\nu op{:}Amb[S])\ (open\ op.\ op[\,]\ |\\
&\qquad c^{ip}[in\ r.\ in\ c^{buf}.\ (x{:}W).\\
&\qquad\quad (\langle x\rangle\ |\ op[out\ c^{buf}.\ out\ r.\ open\ op.\ \langle\!\langle P\rangle\!\rangle])])\\
&\langle\!\langle set\ r\ c\ \langle M_W\rangle.\ P_S\rangle\!\rangle \ \triangleq\\
&\quad (\nu op{:}Amb[S])\ (open\ op.\ op[\,]\ |\\
&\qquad c^{ip}[in\ r.\ in\ c^{buf}.\ (x{:}W).\\
&\qquad\quad (\langle M\rangle\ |\ op[out\ c^{buf}.\ out\ r.\ open\ op.\ \langle\!\langle P\rangle\!\rangle])])
\end{aligned}
$$

The names c^{buf} and c^{ip} related to a cell c are assigned the type $Amb[W]$, where W is the type of the values held by the cell. The name r of a record is simply assigned the type $Amb[Shh]$. A record is able to hold cells of different types.

The type of names of a record field holding W may be denoted by $Field[W]$. This notation can be translated to the ambient calculus by mapping each environment name $n : Field[W]$ to two environment names $n^{buf}, n^{ip} : Amb[W]$, and by mapping each restriction $(\nu n{:}Field[W])\ P$ to the restrictions $(\nu n^{buf}{:}Amb[W])$ $(\nu n^{ip}{:}Amb[W])\ P$.

4.5 Agents

One of the original motivations for the ambient calculus was to provide a natural semantics for wide-area network languages. We now define a simple agent language inspired by Telescript [13]. In the Telescript model, *agents* travel over the network between *places* (agent servers) where agents can meet and communicate with other agents. Agents carry with them a *suitcase* containing local agent data.

The syntax of our stripped-down agent language, Telestrip'd, is described in the following table, together with an informal description of the various constructions. We give the semantics of Telestrip'd by translation to the ambient calculus. The dynamic hierarchical structure of places, agents and suitcases is preserved by our translation; it would not be preserved so obviously by translations into standard process calculi.

We are able to assign types to our definitions, yielding a typed agent language: $Agent[W_1, ..., W_k]$ is the type of names of agents that accept communications of type $W_1\times...\times W_k$.

Telestrip'd Syntax

$W ::= Agent[W_1, ..., W_k]$	agent types ($k \geq 0$)
Net ::=	the network
noplace	no place
place $p[Arena]$	a place called p
$Net\ \|\ Net$	more places
$Arena$ ::=	inside a place
empty	nobody there
agent $(n{:}W)[Code]$	an agent with fresh name n
$Arena\ \|\ Arena$	more agents
$Code$ ::=	agent code
stop	stop
go $p.\ Code$	go to place p and continue
spawn $(n'{:}W)[Code'].\ Code$	spawn a fresh agent n' in the current place
welcome $(n_1{:}W_1, ..., n_k{:}W_k).\ Code$	accept input from a local agent
meet $n\langle n_1, ..., n_k\rangle.\ Code$	output to local agent n
folder $n\ n'.\ Code$	add new folder n with contents n' to the suitcase
get $n(x{:}W).\ Code$	get contents of folder n from the suitcase
set $n\langle n'\rangle.\ Code$	set contents of folder n to n' in the suitcase
...	other constructs (omitted)

Typed Telestrip'd Semantics

$$
\begin{aligned}
&\langle\!\langle Agent[W_1, ..., W_k]\rangle\!\rangle \ \triangleq\ Amb[\langle\!\langle W_1\rangle\!\rangle\times...\times\langle\!\langle W_k\rangle\!\rangle]\\
&\langle\!\langle Net\rangle\!\rangle : Shh\\
&\langle\!\langle Arena\rangle\!\rangle_p : Shh \quad \text{if } p : Amb[Shh]\\
&\langle\!\langle Code\rangle\!\rangle_m : \langle\!\langle W_1\rangle\!\rangle\times...\times\langle\!\langle W_k\rangle\!\rangle \quad \text{if } m : \langle\!\langle Agent[W_1, ..., W_k]\rangle\!\rangle\\[6pt]
&\langle\!\langle noplace\rangle\!\rangle \ \triangleq\ \mathbf{0}\\
&\langle\!\langle place\ p[Arena]\rangle\!\rangle \ \triangleq\ p[\langle\!\langle Arena\rangle\!\rangle_p] \qquad (\text{for } p{:}Amb[Shh])\\
&\langle\!\langle Net\ |\ Net\rangle\!\rangle \ \triangleq\ \langle\!\langle Net\rangle\!\rangle\ |\ \langle\!\langle Net\rangle\!\rangle\\[6pt]
&\langle\!\langle empty\rangle\!\rangle_p \ \triangleq\ \mathbf{0}\\
&\langle\!\langle agent\ (n{:}Agent[W_1, ..., W_k])[Code]\rangle\!\rangle_p \ \triangleq\\
&\quad (\nu n{:}\langle\!\langle Agent[W_1, ..., W_k]\rangle\!\rangle)\\
&\qquad n[record\ sut\ |\ add\ sut\ at\ p\ |\ \langle\!\langle Code\rangle\!\rangle_n]\\
&\langle\!\langle Arena\ |\ Arena\rangle\!\rangle_p \ \triangleq\ \langle\!\langle Arena\rangle\!\rangle_p\ |\ \langle\!\langle Arena\rangle\!\rangle_p\\[6pt]
&\langle\!\langle stop\rangle\!\rangle_m \ \triangleq\ \mathbf{0}\\
&\langle\!\langle go\ p.\ Code\rangle\!\rangle_m \ \triangleq\\
&\quad get\ sut\ at(q{:}Amb[Shh]).\ set\ sut\ at\langle p\rangle.\ out\ q.\ in\ p.\ \langle\!\langle Code\rangle\!\rangle_m\\
&\langle\!\langle spawn\ (n'{:}Agent[W_1, ..., W_k])[Code'].\ Code\rangle\!\rangle_m \ \triangleq\ (\text{for } n' \neq m)\\
&\quad get\ sut\ at(p{:}Amb[Shh]).\ (\nu n', u{:}\langle\!\langle Agent[W_1, ..., W_k]\rangle\!\rangle)\\
&\qquad (n'[record\ sut\ |\ add\ sut\ at\ p\ |\ out\ m.\ open\ u.\ \langle\!\langle Code'\rangle\!\rangle_{n'}]\\
&\qquad\ |\ open\ u.\ \langle\!\langle Code\rangle\!\rangle_m\\
&\qquad\ |\ (\nu t{:}Amb[Shh])\ t[out\ m.\ in\ n'.\ out\ n'.\\
&\qquad\qquad\qquad (u[out\ t.\ in\ n']\ |\ u[out\ t.\ in\ m])])
\end{aligned}
$$

$\langle\!\langle$*meet* $n\langle n_{1W_1}, ..., n_{kW_k}\rangle$. *Code*$\rangle\!\rangle_m$ $\triangleq$
  $(\nu z{:}\langle\!\langle Agent[W_1, ..., W_k]\rangle\!\rangle)$
    $z[$*out m. in n.* $n[$*out z. open z.* $\langle n_1, ..., n_k\rangle]] \mid \langle\!\langle Code\rangle\!\rangle_m$
$\langle\!\langle$*welcome* $(n_1{:}W_1, ..., n_k{:}W_k)$. *Code*$\rangle\!\rangle_m$ $\triangleq$
  open m $\mid (n_1{:}\langle\!\langle W_1\rangle\!\rangle, ..., n_k{:}\langle\!\langle W_k\rangle\!\rangle)$. $\langle\!\langle Code\rangle\!\rangle_m$
$\langle\!\langle$*folder n n'*$_W$. *Code*$\rangle\!\rangle_m$ $\triangleq$
  $(\nu n{:}Field[\langle\!\langle W\rangle\!\rangle])$ (*add sut n n'* $\mid \langle\!\langle Code\rangle\!\rangle_m$)
$\langle\!\langle$*get* $n(x{:}W)$. *Code*$\rangle\!\rangle_m$ $\triangleq$ *get sut* $n(x{:}\langle\!\langle W\rangle\!\rangle)$. $\langle\!\langle Code\rangle\!\rangle_m$
$\langle\!\langle$*set* $n\langle n'\rangle$. *Code*$\rangle\!\rangle_m$ $\triangleq$ *set sut* $n\langle n'\rangle$. $\langle\!\langle Code\rangle\!\rangle_m$
...

No exchange happens at the network level, so the network has type *Shh*. Each arena has also type *Shh*, so the name of each place has type *Amb*[*Shh*].

The type of an ambient reflects only the type of the exchanges performed within it; each agent welcomes (inputs) a single type of data, but can output to agents of several different types. The meet primitive given above is asynchronous; a (more natural) synchronous version is possible but more complicated.

The name of the agent suitcase, *sut*, is a distinguished name of type *Amb*[*Shh*]. A suitcase is a record containing a collection of cells. Each suitcase contains a cell named *at*, a distinguished name of type *Amb*[*Amb*[*Shh*]], containing the name of the agent's current place.

5 Affine Capability Types

In this section, we describe an extension of our type system obtained by adding a new type of affine capabilities $Cap^1[T]$. We enforce the rule that whenever a process inputs a capability of this type, the process may exercise or output the capability at most once.

The motivation for this type system is that in some situations we may want capabilities to play the role of tickets or stamps that may be used once to access a valuable resource (for example, a compute server, or a printer). We would like to guarantee that if a well-typed process is presented with *k* capabilities for accessing a resource, perhaps after a fee has been paid, then that resource is exercised at most *k* times.

5.1 Limiting the Use of Capabilities

Linear type systems for the π-calculus, beginning with the work of Kobayashi, Pierce and Turner [7], restrict the usages of bound names in a variety of ways. Our system is analogous, but is affine (at most one use of names) rather than linear (exactly one use).

We modify the syntax of types by renaming *Cap*[*T*] to $Cap^\omega[T]$ and by introducing a new type, $Cap^1[T]$, of affine capabilities. The *multiplicities* 0, 1 and ω are used to count the number of occurrences of names in terms. We enforce the following simple principles:

- An input name of type $Cap^1[T]$ may be exercised at most once.
- An input name of type $Cap^\omega[T]$ or $Amb[T]$ may be exercised as often as desired, as before.
- A restricted name of type *Amb*[*T*] may be exercised as often as desired, as before.

For example:

- Disallowed: $(x{:}Cap^1[T]).\ (\langle x\rangle \mid \langle x\rangle)$, $(x{:}Cap^1[T]).\ (\langle x\rangle \mid n[x.P])$
- Allowed: $(x{:}Cap^\omega[T]).\ (\langle x\rangle \mid \langle x\rangle \mid n[x.P])$,
  $(x{:}Amb[T]).\ (x[] \mid x[])$, $(x{:}Amb[T]).\ (x[P] \mid x[Q])$,
  $(\nu x{:}Amb[T]).\ (\langle open\ x\rangle \mid \langle open\ x\rangle \mid (y{:}Cap^1[T]).\ \langle y\rangle)$

Here is the syntax of the extended type system:

Types

$W ::=$	message types
$Amb[T]$	ambient name
$Cap^1[T]$	affine capability
$Cap^\omega[T]$	unlimited capability
$T ::=$	exchange types
Shh	no exchange
$W_1\times...\times W_k$	tuple exchange
$\mu ::=$	multiplicities
0	never
1	once
ω	many

We let μ^+ range over {1, ω}.

Let the *multiplicity order*, $\mu \le \mu'$, be the least reflexive and transitive relation to satisfy $0 \le 1 \le \omega$. Let the *addition*, $\mu+\mu'$ of multiplicities μ and μ' be the multiplicity defined by the equations $\mu+0 = 0+\mu = \mu$, $1+1 = \omega$, and $\mu+\omega = \omega+\mu = \omega$. Let the *replication*, $!\mu$, of a multiplicity be the multiplicity $\mu+\mu$.

The functions *n occurs M* and *n occurs P*, given by the following equations, count the occurrences of the name *n* in the term *M* and in the process *P*, respectively. Note that any name under a ! has multiplicity ω.

n occurs m	$\triangleq$	1 if $m=n$; 0 otherwise
n occurs in M	$\triangleq$	*n occurs M*
n occurs out M	$\triangleq$	*n occurs M*
n occurs open M	$\triangleq$	*n occurs M*
n occurs M.M'	$\triangleq$	(*n occurs M*) + (*n occurs M'*)
n occurs ε	$\triangleq$	0
n occurs M.P	$\triangleq$	(*n occurs M*) + (*n occurs P*)
n occurs M[*P*]	$\triangleq$	(*n occurs M*) + (*n occurs P*)
n occurs $(\nu m{:}W)P$	$\triangleq$	(*n occurs M*) for $m \ne n$
n occurs **0**	$\triangleq$	0
n occurs P | *Q*	$\triangleq$	(*n occurs P*) + (*n occurs Q*)
n occurs !*P*	$\triangleq$	!(*n occurs P*)

n occurs $(n_1{:}W_1, ..., n_k{:}W_k).P$ $\triangleq$
  n occurs P for $n \notin \{n_1, ..., n_k\}$
n occurs $\langle M_1, ..., M_k\rangle$ $\triangleq$
  (*n occurs* M_1) + ... + (*n occurs* M_k)

For example:

$$\begin{array}{l}
n\ occurs\ (m[]\ |\ (\nu n{:}W)\ n[]) = 0\\
n\ occurs\ m[n.\mathbf{0}] = 1\\
n\ occurs\ (m[n.\mathbf{0}]\ |\ \langle n\rangle) = \omega
\end{array}$$

We define a new type system using the same rules as before except for the modifications listed below.

Rules

(Exp ε)
$$\frac{E \vdash \diamond}{E \vdash \varepsilon : Cap^{\mu^+}[T]}$$

(Exp .)
$$\frac{E \vdash M : Cap^{\mu^+}[T] \qquad E \vdash M' : Cap^{\mu^+}[T]}{E \vdash M.M' : Cap^{\mu^+}[T]}$$

(Exp In)
$$\frac{E \vdash M : Amb[S]}{E \vdash in\ M : Cap^{\mu^+}[T]}$$

(Exp Out)
$$\frac{E \vdash M : Amb[S]}{E \vdash out\ M : Cap^{\mu^+}[T]}$$

(Exp Open)
$$\frac{E \vdash M : Amb[T]}{E \vdash open\ M : Cap^{\mu^+}[T]}$$

(Proc Action)
$$\frac{E \vdash M : Cap^{\mu^+}[T] \qquad E \vdash P : T}{E \vdash M.P : T}$$

(Proc Input) (where $\forall i \in 1..k.\ W_i = Cap^1[T_i] \Rightarrow n_i\ occurs\ P \le 1$)
$$\frac{E, n_1{:}W_1, ..., n_k{:}W_k \vdash P : W_1 \times ... \times W_k}{E \vdash (n_1{:}W_1, ..., n_k{:}W_k).P : W_1 \times ... \times W_k}$$

A subject reduction result can be proven for the modified system (the proof is in Appendix 8).

5-1 Proposition (Subject Reduction)

If $E \vdash P : U$ and $P \longrightarrow Q$ then $E \vdash Q : U$.

☐

5.2 Avoiding a Synchronization Error Using Affine Types

To illustrate the use of affine capability types, we describe a taxi protocol. This protocol uses affine typing to achieve proper movement synchronization between two parties. The taxi publishes a capability for a passenger to enter a seat in the back of the taxi. The passenger enters and tells the taxi a route to follow. At the end of the trip the taxi door is unlocked, and the passenger may exit. The capabilities for entering and exiting the taxi, and for the route, are given affine types.

If the capability to enter the taxi were to be accidentally or maliciously duplicated, a synchronization error could arise, in which a passenger holding a valid capability would attempt to enter the taxi, but would be left behind because another passenger got the taxi first. This possibility is ruled out by affine typing.

In the following, the parameter M is the route the taxi is to follow, and the parameter P is the behavior of the passenger at the destination.

$$\begin{array}{l}
passenger\ M\ P \ \triangleq\\
\quad (enter{:}Cap^1[Shh]).\\
\qquad move[enter.\ talk[out\ move.\ \langle M\rangle\ |\\
\qquad\quad talk[(exit{:}Cap^1[Shh]).\ move[exit.\ P]]]]\\
taxi \ \triangleq\\
\quad (\nu\ taxi{:}Amb[Shh],\ go{:}Amb[Shh],\\
\qquad lock{:}Amb[Shh],\ seat{:}Amb[Cap^1[Shh]])\\
\qquad\quad (\langle in\ taxi.\ in\ lock.\ in\ seat\rangle\ |\\
\qquad\quad\ taxi[open\ go\ |\\
\qquad\qquad lock[\\
\qquad\qquad\quad seat[open\ talk.\ (route{:}Cap^1[Shh]).\\
\qquad\qquad\qquad (open\ talk.\ \langle out\ seat.\ out\ taxi\rangle\ |\\
\qquad\qquad\qquad\ go[out\ seat.\ out\ lock.\ route.\ open\ lock])]]]])
\end{array}$$

If we suppose there is some environment E with:

$$\begin{array}{ll}
E \vdash talk : Amb[Cap^1[Shh]] & E \vdash M : Cap^1[Shh]\\
E \vdash move : Amb[Shh] & E \vdash P : Shh
\end{array}$$

then:

$$E \vdash (passenger\ M\ P\ |\ taxi) : Cap^1[Shh]$$

(N.B.: the passenger-taxi system can also be given type $Cap^{\omega}[Shh]$. We can force the $Cap^1[Shh]$ typing by situating the system within an ambient whose name has type $Amb[Cap^1[Shh]]$.)

Initially, the system reduces as follows, up to the point where the passenger has entered the taxi and the taxi is ready to follow the route M:

$$\begin{array}{l}
passenger\ M\ P\ |\ taxi \ \longrightarrow^*\\
\quad (\nu\ taxi{:}Amb[Shh],\ lock{:}Amb[Shh],\ seat{:}Amb[Cap^1[Shh]])\\
\qquad taxi[M.\ open\ lock\ |\\
\qquad\quad lock[move[out\ taxi.\ P]\ |\ seat[move[]]]]
\end{array}$$

Once the route M has been followed, the lock ambient is opened, and the passenger exits.

5.3 Dispensing Transferrable Tokens Using Affine Types

A second example demonstrates that affine types allow capabilities to serve as consumable, transferrable tokens for a resource.

We consider a system consisting of several principals that are given access to a printer. Each principal has an API (interface) allowing it to print messages on the printer. Each time it accesses the API, a principal must consume a token, the capability *open api*. This capability is given the affine type $Cap^1[Msg]$, where Msg is the type of messages printed by the printer. By dispensing different numbers of these tokens to different principals, we may selectively control the number of messages each principal has a right to print. The top-level of our system, *sys*, serves as a printer spool; any outputs here may be thought of as being sent to a printer.

We describe each principal as follows:

$$principal\ n\ P \triangleq$$
$$n[open\ n \mid printAPI\ n \mid toks[!open\ toks \mid P]]$$

The process parameter *P* models the specific behavior of the principal. We assume that the names *api* and *print* are not free in *P*. The ambient named *toks* represents a channel on which the principal receives capabilities for printing. A token *open api* provides access to the printer API, which is defined by:

$$printAPI\ n \triangleq\ !api[(x{:}Msg).\ print[out\ n.\ \langle x \rangle]]$$

Our example system consists of two principals, named *alice* and *bob*:

$$\begin{array}{l} sys \triangleq \\ \quad (\nu\ alice{:}Amb[Msg],\ bob{:}Amb[Msg],\ api{:}Amb[Msg], \\ \qquad print{:}Amb[Msg],\ toks{:}Amb[Cap^1[Msg]]) \\ \qquad\ (!open\ print \mid \\ \qquad\quad toks[in\ alice.\ in\ toks \mid \langle open\ api \rangle \mid \langle open\ api \rangle] \mid \\ \qquad\quad principal\ alice\ ((x_1{:}Cap^1[Msg]).\ (x_2{:}Cap^1[Msg]). \\ \qquad\qquad alice[out\ toks \mid x_1.\ \langle M \rangle \mid \\ \qquad\qquad\quad toks[out\ alice.\ in\ bob.\ in\ toks.\ \langle x_2 \rangle]]) \\ \qquad\quad principal\ bob\ ((y{:}Cap^1[Msg]). \\ \qquad\qquad bob[out\ toks.\ y.\ \langle N \rangle])) \end{array}$$

In this example, we dispense two tokens to *alice*, via the process *toks*[*in alice*. *in toks* | ⟨*open api*⟩ | ⟨*open api*⟩], but none to *bob*. Principal *alice* inputs the two tokens as variables x_1 and x_2; she uses x_1 herself to print *M*, but donates the other to *bob*, who inputs it as *y*, and uses it to print *N*.

The process *sys* has type *Msg*. We have:

$$sys \longrightarrow^* \langle M \rangle \mid \langle N \rangle$$

We may easily add more principals to this example, and we may dispense as many tokens as is appropriate to each new principal. By using affine types to regulate the use of printer tokens, principals are free to transfer tokens amongst themselves, but the total number of messages printed is limited by the number of tokens dispensed initially. Without linear or affine types, it would be harder to allow the transfer of printer tokens between principals while still controlling their total number.

6 Conclusions

We have presented a type system for the ambient calculus. The types arising from this work are unusual in that they do not correspond directly to channel or function types. The type system guarantees the soundness of message exchanges, while leaving great flexibility in mobility. As an example, we have given a natural semantics for a typed agent language.

Our type system is rather basic, roughly corresponding to the simply-typed discipline for the λ-calculus. Much richer typing disciplines can be imagined, along the usual lines. Perhaps more interestingly, it is appealing to try and use static type systems to restrict mobility; this is the subject of current work.

Acknowledgments

Georges Gonthier made useful remarks on an early draft, and discovered new π-calculus encodings that illustrate interesting techniques.

7 Appendix: Subject Reduction

Let $E \vdash J$ denote any judgment.

7-1 Lemma
If $E', n{:}W, E'' \vdash J$ then $n \notin dom(E', E'')$.

7-2 Lemma
If $E \vdash n : W$ and $E \vdash n : W'$, then $W{=}W'$.

7-3 Lemma (Implied Judgment)
If $E', E'' \vdash J$ then $E' \vdash \diamond$.

7-4 Lemma (Exchange)
If $E', n{:}W', m{:}W'', E'' \vdash J$ then $E', m{:}W'', n{:}W', E'' \vdash J$.

7-5 Lemma (Weakening)
If $E', E'' \vdash J$ and $n \notin dom(E', E'')$ then $E', n{:}W, E'' \vdash J$.

7-6 Lemma (Strengthening)
If $E', n{:}W, E'' \vdash J$ and $n \notin fn(J)$ then $E', E'' \vdash J$.

7-7 Lemma (Substitution)
If $E', n{:}W, E'' \vdash J$ and $E' \vdash M : W$ then $E', E'' \vdash J\{n{\leftarrow}M\}$.

7-8 Proposition (Subject Congruence)
(1) If $E \vdash P : U$ and $P \equiv Q$ then $E \vdash Q : U$.
(2) If $E \vdash P : U$ and $Q \equiv P$ then $E \vdash Q : U$.

Proof

By mutual induction on the derivations of $P \equiv Q$ and $Q \equiv P$.

(1) If $E \vdash P : U$ and $P \equiv Q$ then $E \vdash Q : U$.

(Struct Refl) Trivial.

(Struct Symm) Then $Q \equiv P$. By induction hypothesis (2), we have $E \vdash Q : U$.

(Struct Trans) Then $P \equiv R$, $R \equiv Q$ for some *R*. By induction hypothesis (1), $E \vdash R : U$. Again by induction hypothesis (1), $E \vdash Q : U$.

(Struct Res) Then $P = (\nu n{:}W)P'$ and $Q = (\nu n{:}W)Q'$, with $P' \equiv Q'$. Assume $E \vdash P : U$. This must have been derived from (Proc Res), with $E, n{:}Amb[T] \vdash P' : U$, where $W{=}Amb[T]$. By induction hypothesis, $E, n{:}Amb[T] \vdash Q' : U$. By (Proc Res) $E \vdash (\nu n{:}Amb[T])Q' : U$.

(Struct Par) Then $P = P' \mid R$, $Q = Q' \mid R$, and $P' \equiv Q'$. Assume $E \vdash P' \mid R : U$. This must have been derived from (Proc Par), with $E \vdash P' : U$ and $E \vdash R : U$. By induction hypothesis $E \vdash Q' : U$. By (Proc Par) $E \vdash Q' \mid R : U$.

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

(Struct Amb) Then $P = M[P']$, $Q = M[Q']$, and $P' \equiv Q'$. Assume $E \vdash P : U$. This must have been derived from (Proc Amb), with $E \vdash M : Amb[T]$ and $E \vdash P' : T$ for some T. By induction hypothesis, $E \vdash Q' : T$. By (Proc Amb) we derive $E \vdash M[Q'] : U$.

(Struct Action) Then $P = M.P'$, $Q = M.Q'$, and $P' \equiv Q'$. Assume $E \vdash P : U$. This must have been derived from (Proc Action), with $E \vdash M : Cap[U]$ and $E \vdash P' : U$. By induction hypothesis, $E \vdash Q' : U$. By (Proc Action) $E \vdash M[Q'] : U$.

(Struct Input) Then $P = (n_1{:}W_1, ..., n_k{:}W_k).P'$, $Q = (n_1{:}W_1, ..., n_k{:}W_k).Q'$, and $P' \equiv Q'$. Assume $E \vdash P : U$. This must have been derived from (Proc Input), with $E, n_1{:}W_1, ..., n_k{:}W_k \vdash P' : U$, where $U = W_1\times...\times W_k$. By induction hypothesis, $E, n_1{:}W_1, ..., n_k{:}W_k \vdash Q' : U$. By (Proc Input) $E \vdash (n_1{:}W_1, ..., n_k{:}W_k).Q' : U$.

(Struct Par Comm) Then $P = P' \mid P''$ and $Q = P'' \mid P'$. Assume $E \vdash P' \mid P'' : U$. This must have been derived from (Proc Par), with $E \vdash P' : U$ and $E \vdash P'' : U$. By (Proc Par) $E \vdash P'' \mid P' : U$.

(Struct Par Assoc) Then $P = (P' \mid P'') \mid P'''$ and $Q = P' \mid (P'' \mid P''')$. Assume $E \vdash (P' \mid P'') \mid P''' : U$. This must have been derived from (Proc Par) twice, with $E \vdash P' : U$, $E \vdash P'' : U$, and $E \vdash P''' : U$. By (Proc Par) twice, $E \vdash P' \mid (P'' \mid P''') : U$.

(Struct Repl Par) Then $P = !P'$ and $Q = P' \mid !P'$. Assume $E \vdash !P' : U$. This must have been derived from (Proc Repl), with $E \vdash P' : U$. By (Proc Par), $E \vdash P' \mid !P' : U$.

(Struct Res Res) Then $P = (\nu n{:}W)(\nu m{:}V)P'$ and $Q = (\nu m{:}V)(\nu n{:}W)P'$ with $n \neq m$. Assume $E \vdash (\nu n{:}W)(\nu m{:}V)P' : U$. This must have been derived from (Proc Res) twice, with $E, n{:}Amb[T], m{:}Amb[S] \vdash P' : U$, where $W{=}Amb[T]$ and $V{=}Amb[S]$. By Lemma 7-4 we have $E, m{:}Amb[S], n{:}Amb[T] \vdash P' : U$. By (Proc Res) twice we have $E \vdash (\nu m{:}Amb[S])(\nu n{:}Amb[T])P' : U$.

(Struct Res Par) Then $P = (\nu n{:}W)(P' \mid P'')$ and $Q = P' \mid (\nu n{:}W)P''$, with $n \notin fn(P')$. Assume $E \vdash P : U$. This must have been derived from (Proc Res), with $E, n{:}Amb[T] \vdash P' \mid P'' : U$ and $W = Amb[T]$, and from (Proc Par), with $E, n{:}Amb[T] \vdash P' : U$ and $E, n{:}Amb[T] \vdash P'' : U$. By Lemma 7-6, since $n \notin fn(P')$, we have $E \vdash P' : U$. By (Proc Res) we have $E \vdash (\nu n{:}Amb[T])P'' : U$. By (Proc Par) we have $E \vdash P' \mid (\nu n{:}Amb[T])P'' : U$.

(Struct Res Amb) Then $P = (\nu n{:}W)m[P']$ and $Q = m[(\nu n{:}W)P']$, with $n \neq m$. Assume $E \vdash P : U$. This must have been derived from (Proc Res) with $E{,}n{:}Amb[T] \vdash m[P'] : U$ with $W = Amb[T]$, and from (Proc Amb) with $E, n{:}Amb[T] \vdash P' : S$ and $E, n{:}Amb[T] \vdash m : Amb[S]$ for some S. By (Proc Res) we have $E \vdash (\nu n{:}Amb[T])P' : S$. By Lemma 7-6, since $n \neq m$, we have $E \vdash m : Amb[S]$. By (Proc Amb) we can derive $E \vdash m[(\nu n{:}Amb[T])P'] : U$.

(Struct Zero Par) Then $P = P' \mid \mathbf{0}$ and $Q = P'$. Assume $E \vdash P : U$. This must have been derived from (Proc Par) with $E \vdash P' : U$ and $E \vdash \mathbf{0} : U$.

(Struct Zero Res) Then $P = (\nu n{:}W)\mathbf{0}$ and $Q = \mathbf{0}$. Assume $E \vdash P : U$. This must have been derived from (Proc Res) with $E, n{:}Amb[T] \vdash \mathbf{0} : U$ and $W = Amb[T]$. By Lemma 7-6, $E \vdash \mathbf{0} : U$.

(Struct Zero Repl) Then $P = !\mathbf{0}$ and $Q = \mathbf{0}$. Assume $E \vdash P : U$. This must have been derived from (Proc Repl) with $E \vdash \mathbf{0} : U$.

(Struct ε) Then $P = \varepsilon.P'$ and $Q = P'$. Assume $E \vdash P : U$. This must have been derived from (Proc Action) with $E \vdash P' : U$.

(Struct .) Then $P = (M.M').P'$ and $Q = M.M'.P'$. Assume $E \vdash P : U$. This must have been derived from (Proc Action) with $E \vdash M.M' : Cap[U]$ and $E \vdash P' : U$. The former must come from (Exp .) with $E \vdash M : Cap[U]$ and $E \vdash M' : Cap[U]$. By (Proc Action) twice we have $E \vdash M.M'.P' : U$.

(2) If $E \vdash P : U$ and $Q \equiv P$ then $E \vdash Q : U$.

(Struct Refl) Trivial.

(Struct Symm) Then $P \equiv Q$. By induction hypothesis (1), we have $E \vdash Q : U$.

(Struct Trans) Then $Q \equiv R$, $R \equiv P$ for some R. By induction hypothesis (2), $E \vdash R : U$ and $E \vdash Q : U$.

(Struct Res), (Struct Par), (Struct Repl), (Struct Amb), (Struct Action), (Struct Input), (Struct Par Assoc) Symmetrical to case (1).

(Struct Par Comm) Then $Q = P' \mid P''$ and $P = P'' \mid P'$. Assume $E \vdash P'' \mid P' : U$. This must have been derived from (Proc Par), with $E \vdash P'' : U$ and $E \vdash P' : U$. By (Proc Par) $E \vdash P' \mid P'' : U$.

(Struct Repl Par) Then $Q = !P'$ and $P = P' \mid !P'$. Assume $E \vdash P' \mid !P' : U$. This must have been derived from (Proc Par), with $E \vdash !P' : U$.

(Struct Res Res) Then $Q = (\nu n{:}W)(\nu m{:}V)P'$ and $P = (\nu m{:}V)(\nu n{:}W)P'$ with $n \neq m$. Assume $E \vdash (\nu m{:}V)(\nu n{:}W)P' : U$. This must have been derived from (Proc Res) twice, with $E, m{:}Amb[S], n{:}Amb[T] \vdash P' : U$, where $W{=}Amb[T]$ and $V{=}Amb[S]$. By Lemma 7-4 we have $E, n{:}Amb[T], m{:}Amb[S] \vdash P' : U$. By (Proc Res) twice we have $E \vdash (\nu n{:}Amb[T])(\nu m{:}Amb[S])P' : U$.

(Struct Res Par) Then $Q = (\nu n{:}W)(P' \mid P'')$ and $P = P' \mid (\nu n{:}W)P''$, with $n \notin fn(P')$. Assume $E \vdash P : U$. This must have been derived from (Proc Par), with $E \vdash P' : U$ and $E \vdash (\nu n{:}W)P'' : U$, and the latter from (Proc Res), with $E, n{:}Amb[T] \vdash P'' : U$ where $W{=}Amb[T]$. By Lemma 7-5, since $n \notin dom(E')$, we have $E, n{:}Amb[T] \vdash P' : U$. By (Proc Par) we have $E, n{:}Amb[T] \vdash P' \mid P'' : U$. By (Proc Res) we have $E \vdash (\nu n{:}Amb[T])(P' \mid P'') : U$.

(Struct Res Amb) Then $Q = (\nu n{:}W)m[P']$ and $P = m[(\nu n{:}W)P']$, with $n \neq m$. Assume $E \vdash P : U$. This must have been derived from (Proc Amb) with $E \vdash m : Amb[S]$ and $E \vdash (\nu n{:}W)P' : S$ for some S. The latter must have been derived from (Proc Res) with $E, n{:}Amb[T] \vdash P' : S$ with $W = Amb[T]$. By Lemma 7-5, since $n \notin dom(E)$, we have $E, n{:}Amb[T] \vdash m : Amb[S]$. By (Proc Amb) we can derive $E, n{:}Amb[T] \vdash m[P'] : U$. By (Proc Res) we have $E \vdash (\nu n{:}Amb[T])m[P'] : U$.

(Struct Zero Par) Then $Q = P' \mid \mathbf{0}$ and $P = P'$. Assume $E \vdash P : U$. By Lemma 7-3, $E \vdash \Diamond$. By (Proc Zero) $E \vdash \mathbf{0} : U$. By (Proc Par), $E \vdash P' \mid \mathbf{0} : U$.

(Struct Zero Res) Then $Q = (\nu n{:}Amb[T])\mathbf{0}$ and $P = \mathbf{0}$. Assume $E \vdash P : U$. By Lemma 7-5, $E, n{:}Amb[T] \vdash \mathbf{0} : U$. By (Proc Res) $E \vdash (\nu n{:}Amb[T])\mathbf{0} : U$.

(Struct Zero Repl) Then $Q = !\mathbf{0}$ and $P = \mathbf{0}$. Assume $E \vdash P : U$. By (Proc Repl) with $E \vdash !\mathbf{0} : U$.

(Struct ε) Then $Q = \varepsilon.P'$ and $P = P'$. Assume $E \vdash P : U$. By Lemma 7-3, $E \vdash \diamond$. By (Exp ε), $E \vdash \varepsilon : Cap[U]$. By (Proc Action) with $E \vdash \varepsilon.P' : U$.

(Struct .) Then $Q = (M.M').P'$ and $P = M.M'.P'$. Assume $E \vdash P : U$. This must have been derived from (Proc Action) twice, with $E \vdash M : Cap[U]$, $E \vdash M' : Cap[U]$, and $E \vdash P' : U$. By (Exp .) we have $E \vdash M.M' : Cap[U]$. By (Proc Action) we have $E \vdash (M.M').P' : U$.

□

7-9 Proof of Proposition 3-1 (Subject Reduction)

If $E \vdash P : U$ and $P \longrightarrow Q$ then $E \vdash Q : U$.

Proof

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

(Red In) Then $P = n[in\ m.\ P' \mid P''] \mid m[P''']$ and $Q = m[n[P' \mid P''] \mid P''']$. Assume $E \vdash P : U$. This must have been derived from (Proc Par), with $E \vdash n[in\ m.\ P' \mid P''] : U$ and $E \vdash m[P'''] : U$. Those two judgments must have been derived from (Proc Amb), with $E \vdash n : Amb[T]$, $E \vdash in\ m.\ P' \mid P'' : T$ for some T, and $E \vdash m : Amb[S]$, $E \vdash P''' : S$ for some S. Moreover, $E \vdash in\ m.\ P' \mid P'' : T$ must come from (Proc Par) with $E \vdash in\ m.\ P' : T$ and $E \vdash P'' : T$, and $E \vdash in\ m.\ P' : T$ must come from (Proc Action) with $E \vdash in\ m : Cap[T]$ and $E \vdash P' : T$. Note that $E \vdash m : Amb[S]$ is consistent with $E \vdash in\ m : Cap[T]$, by (Exp In). By (Proc Par) we have $E \vdash P' \mid P'' : T$, and by (Proc Amb) we can derive $E \vdash n[P' \mid P''] : S$. Then, by (Proc Par) we have $E \vdash n[P' \mid P''] \mid P''' : S$, and by (Proc Amb) we can derive $E \vdash m[n[P' \mid P''] \mid P'''] : U$.

(Red Out) Then $P = m[n[out\ m.\ P' \mid P''] \mid P''']$ and $Q = n[P' \mid P''] \mid m[P''']$. Assume $E \vdash P : U$. This must have been derived from (Proc Amb), with $E \vdash m : Amb[T]$ and $E \vdash n[out\ m.\ P' \mid P''] \mid P''' : T$ for some T. The latter must come from (Proc Par) with $E \vdash P''' : T$ and $E \vdash n[out\ m.\ P' \mid P''] : T$. The latter must come from (Proc Amb) with $E \vdash n : Amb[S]$ and $E \vdash out\ m.\ P' \mid P'' : S$ for some S. The latter must come from (Proc Par) with $E \vdash P'' : S$ and $E \vdash out\ m.\ P' : S$. The latter must come from (Proc Action) with $E \vdash out\ m : Cap[S]$ and $E \vdash P' : S$. Note that $E \vdash m : Amb[T]$ is consistent with $E \vdash out\ m : Cap[S]$, by (Exp Out). By (Proc Par) we have $E \vdash P' \mid P'' : S$, and by (Proc Amb) we can derive $E \vdash n[P' \mid P''] : U$. Then, by (Proc Amb) we can derive $E \vdash m[P'''] : U$, and by (Proc Par) we have $E \vdash n[P' \mid P''] \mid m[P'''] : U$.

(Red Open) Then $P = open\ n.\ P' \mid n[P'']$ and $Q = P' \mid P''$. Assume $E \vdash P : U$. This must have been derived from (Proc Par), with $E \vdash open\ n.\ P' : U$ and $E \vdash n[P''] : U$. The judgment $E \vdash open\ n.\ P' : U$ must have been derived from (Proc Action), with $E \vdash open\ n : Cap[U]$ and $E \vdash P' : U$, and from (Exp Open) with $E \vdash n : Amb[U]$. The judgment $E \vdash n[P''] : U$ must have then been derived from (Proc Amb) with $E \vdash n : Amb[U']$, and $E \vdash P'' : U'$. By Lemma 7-2, $U'=U$. By (Proc Par) we have $E \vdash P' \mid P'' : U$.

(Red Comm) Then $P = (n_1{:}W_1, ..., n_k{:}W_k).P' \mid \langle M_1, ..., M_k \rangle$ and $Q = P'\{n_1{\leftarrow}M_1, ..., n_k{\leftarrow}M_k\}$. Assume $E \vdash P : U$. This must have been derived from (Proc Par) with $E \vdash (n_1{:}W_1, ..., n_k{:}W_k).P' : U$ and $E \vdash \langle M_1, ..., M_k \rangle : U$. The former must have been derived from (Proc Input) with $E, n_1{:}W_1, ..., n_k{:}W_k \vdash P' : W_1{\times}...{\times}W_k$, and $U = W_1{\times}...{\times}W_k$. The latter must have been derived from (Proc Output) with $E \vdash M_1 : W_1'$ ... $E \vdash M_k : W_k'$, and $U = W_1'{\times}...{\times}W_k'$. Hence, $W_1 = W_1'$ ... $W_k = W_k'$. By k applications of Lemma 7-7, we have that $E \vdash P'\{n_1{\leftarrow}M_1, ..., n_k{\leftarrow}M_k\} : U$.

(Red Res) Then $P = (\nu n{:}W)P'$, $Q = (\nu n{:}W)Q'$, and $P' \longrightarrow Q'$. Assume $E \vdash P : U$. This must have been derived from (Proc Res) with $E, n{:}Amb[T] \vdash P' : U$, where $W = Amb[T]$. By induction hypothesis $E, n{:}Amb[T] \vdash Q' : U$. By (Proc Res), $E \vdash (\nu n{:}Amb[T])Q' : U$.

(Red Amb) Then $P = n[P']$, $Q = n[Q']$, and $P' \longrightarrow Q'$. Assume $E \vdash P : U$. This must have been derived from (Proc Amb) with $E \vdash n : Amb[T]$ and $E \vdash P' : T$ for some T. By induction hypothesis, $E \vdash Q' : T$. Then, by (Proc Amb) we can derive $E \vdash n[Q'] : U$.

(Red Par) Then $P = P' \mid R$, $Q = Q' \mid R$, and $P' \longrightarrow Q'$. Assume $E \vdash P : U$. This must have been derived from (Proc Par) with $E \vdash P' : U$ and $E \vdash R : U$. By induction hypothesis, $E \vdash Q' : U$. By (Proc Par) $E \vdash Q' \mid R : U$.

(Red ≡) Then $P \equiv P'$, $Q \equiv Q'$, and $P' \longrightarrow Q'$. Assume $E \vdash P : U$. By Proposition 7-8, $E \vdash P' : U$. By induction hypothesis, $E \vdash Q' : U$. By Proposition 7-8, $E \vdash Q : U$.

□

8 Appendix: Subject Reduction for Affine Types

8-1 Lemma

$\mu+\mu' = \mu'+\mu$

8-2 Lemma

$(\mu+\mu')+\mu'' = \mu+(\mu'+\mu'')$

8-3 Lemma

$!\mu = \mu + !\mu$

8-4 Lemma

If $P \equiv Q$ then $n\ occurs\ P = n\ occurs\ Q$.

Proof

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

(Struct Refl) Trivial.

(Struct Symm) Then $Q \equiv P$. By induction hypothesis, we have $n\ occurs\ Q = n\ occurs\ P$.

(Struct Trans) Then $P \equiv R$, $R \equiv Q$ for some R. By induction hypothesis, $n\ occurs\ P = n\ occurs\ R$. Again by induction hypothesis, $n\ occurs\ R = n\ occurs\ Q$. Hence, $n\ occurs\ P = n\ occurs\ Q$.

(Struct Res) Then $P = (\nu m{:}W)P'$ and $Q = (\nu m{:}W)Q'$, with $P' \equiv Q'$. Since m is bound, we may assume that $m \neq n$. By induction hypothesis, $n\ occurs\ P' = n\ occurs\ Q'$. Therefore, $n\ occurs\ P = n\ occurs\ P' = n\ occurs\ Q' = n\ occurs\ Q$.

(Struct Par) Then $P = P' \mid R$, $Q = Q' \mid R$, and $P' \equiv Q'$. By induction hypothesis, $n\ occurs\ P' = n\ occurs\ Q'$. Therefore, $n\ occurs\ P = (n\ occurs\ P') + (n\ occurs\ R) = (n\ occurs\ Q') + (n\ occurs\ R) = n\ occurs\ Q$.

(Struct Repl) Then $P = !P'$, $Q = !Q'$, and $P' \equiv Q'$. By induction hypothesis, *n occurs P' = n occurs Q'*. Therefore, *n occurs P* = !(*n occurs P'*) = !(*n occurs Q'*) = *n occurs Q*.

(Struct Amb) Then $P = M[P']$, $Q = M[Q']$, and $P' \equiv Q'$. By induction hypothesis, *n occurs P' = n occurs Q'*. Therefore, *n occurs P* = (*n occurs M*) + (*n occurs P'*) = (*n occurs M*) + (*n occurs Q'*) = *n occurs Q*.

(Struct Action) Then $P = M.P'$, $Q = M.Q'$, and $P' \equiv Q'$. By induction hypothesis, *n occurs P' = n occurs Q'*. Therefore, *n occurs P* = (*n occurs M*) + (*n occurs P'*) = (*n occurs M*) + (*n occurs Q'*) = *n occurs Q*.

(Struct Input) Then $P = (n_1{:}W_1, ..., n_k{:}W_k).P'$, $Q = (n_1{:}W_1, ..., n_k{:}W_k).Q'$, and $P' \equiv Q'$. Since the names $n_1, ..., n_k$ are bound, we may assume that $n \notin \{n_1, ..., n_k\}$. By induction hypothesis, *n occurs P' = n occurs Q'*. Therefore, *n occurs P = n occurs P' = n occurs Q' = n occurs Q*.

(Struct Par Comm) Then $P = P' \mid P''$ and $Q = P'' \mid P'$. By Lemma 8-1, we have: *n occurs P* = (*n occurs P'*) + (*n occurs P"*) = (*n occurs P"*)+(*n occurs P'*) = *n occurs Q*.

(Struct Par Assoc) Then $P = (P' \mid P'') \mid P'''$ and $Q = P' \mid (P'' \mid P''')$. By Lemma 8-2, we have: *n occurs P* = ((*n occurs P'*) + (*n occurs P"*)) + (*n occurs P'"*) = (*n occurs P'*)+((*n occurs P"*)+(*n occurs P'"*)) = *n occurs Q*.

(Struct Repl Par) Then $P = !P'$ and $Q = P' \mid !P'$. By Lemma 8-3, we have: *n occurs P* = !(*n occurs P'*) = (*n occurs P'*) + !(*n occurs P'*) = *n occurs Q*.

(Struct Res Res) Then $P = (\nu m{:}W)(\nu m'{:}V)P'$ and $Q = (\nu m'{:}V)(\nu m{:}W)P'$ with $m \neq m'$. Since the names m and m' are bound, we may assume that $n \neq m$ and $n \neq m'$. Therefore, *n occurs P = n occurs P' = n occurs Q*.

(Struct Res Par) Then $P = (\nu m{:}W)(P' \mid P'')$ and $Q = P' \mid (\nu m{:}W)P''$, with $m \notin fn(P')$. Since the name m is bound, we may assume that $n \neq m$. Therefore, *n occurs P* = (*n occurs P'*) + (*n occurs P"*) = *n occurs Q*.

(Struct Res Amb) Then $P = (\nu m{:}W)m'[P']$ and $Q = m'[(\nu m{:}W)P']$, with $m \neq m'$. Since the name m is bound, we may assume that $n \neq m$. Therefore, *n occurs P* = (*n occurs m'*)+(*n occurs P'*) = *n occurs Q*.

(Struct Zero Par) Then $P = P' \mid \mathbf{0}$ and $Q = P'$. We have: *n occurs P* = (*n occurs P'*) + 0 = *n occurs P' = n occurs Q*.

(Struct Zero Res) Then $P = (\nu m{:}W)\mathbf{0}$ and $Q = \mathbf{0}$. We have, *n occurs P = n occurs Q*.

(Struct Zero Repl) Then $P = !\mathbf{0}$ and $Q = \mathbf{0}$. We have *n occurs P* = 0 = *n occurs Q*.

(Struct ε) Then $P = \varepsilon.P'$ and $Q = P'$. We have *n occurs P = n occurs P' = n occurs Q*.

(Struct .) Then $P = (M.M').P'$ and $Q = M.M'.P'$. By Lemma 8-2, we have *n occurs P* = ((*n occurs M*) + (*n occurs M'*)) + *n occurs P'* = *n occurs M* + (*n occurs M'* + *n occurs P'*) = *n occurs Q*.

□

8-5 Lemma

If $n \notin fn(M)$ then *n occurs M* = 0.

Proof

By induction on the structure of M.

□

8-6 Lemma

If $n \notin \{m\} \cup fn(M)$ then:

(1) *n occurs* $N\{m\leftarrow M\}$ = *n occurs N*.

(2) *n occurs* $P\{m\leftarrow M\}$ = *n occurs P*.

Proof

By inductions on the structure of N and P.

□

The extended type system is as follows: the judgments are as in Section 3.3, and the rules are as in Section 3.3, except for the modifications described in Section 5. We now prove subject reduction for the extended system.

8-7 Lemma

If $E \vdash M : T$ then $fn(M) \subseteq dom(E)$.

8-8 Lemma

If $E', n{:}W, E'' \vdash J$ then $n \notin dom(E',E'')$.

8-9 Lemma

If $E \vdash n : W$ and $E \vdash n : W'$, then $W = W'$.

8-10 Lemma (Implied Judgment)

If $E', E'' \vdash J$ then $E' \vdash \lozenge$.

8-11 Lemma (Exchange)

If $E', n{:}W', m{:}W'', E'' \vdash I$ then $E', m{:}W'', n{:}W', E'' \vdash J$.

8-12 Lemma (Weakening)

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

8-13 Lemma (Strengthening)

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

8-14 Lemma (Substitution)

If $E', n{:}W, E'' \vdash J$ and $E' \vdash M : W$ then $E', E'' \vdash J\{n\leftarrow M\}$.

Proof

By induction on the derivation of $E', n{:}W, E'' \vdash J$.

(Proc Input) We have $E', n{:}W, E'' \vdash (n_1{:}W_1, ..., n_k{:}W_k).P : T$ derived from $E', n{:}W, E'', n_1{:}W_1, ..., n_k{:}W_k \vdash P : T$ and $T = W_1 \times ... \times W_k$. Moreover, for all $i \in 1..k$, if $W_i = Cap^1[T_i]$ then n_i *occurs P* ≤ 1. By induction hypothesis, $E', E'', n_1{:}W_1, ..., n_k{:}W_k \vdash P\{n\leftarrow M\} : T$. By Lemma 8-7, $fn(M) \subseteq dom(E')$. Hence, by Lemma 8-8, $(\{n\} \cup fn(M)) \cap \{n_1, ..., n_k\} = \emptyset$. By Lemma 8-6, n_i *occurs* $P\{n\leftarrow M\}$) = n_i *occurs P*, for all $i \in 1..k$. By (Proc Input), $E', E'' \vdash (n_1{:}W_1, ..., n_k{:}W_k).(P\{n\leftarrow M\}) : T$. Since $(\{n\} \cup fn(M)) \cap \{n_1, ..., n_k\} = \emptyset$, this is to say that $E', E'' \vdash ((n_1{:}W_1, ..., n_k{:}W_k).P)\{n\leftarrow M\}) : T$.

Other cases. The other cases are almost exactly as before.

□

8-15 Proposition (Subject Congruence)

(1) If $E \vdash P : U$ and $P \equiv Q$ then $E \vdash Q : U$.

(2) If $E \vdash P : U$ and $Q \equiv P$ then $E \vdash Q : U$.

Proof

By mutual inductions on derivations.

(Struct Input) Then $P = (n_1{:}W_1, ..., n_k{:}W_k).P'$, $P' \equiv Q'$, and $Q = (n_1{:}W_1, ..., n_k{:}W_k).Q'$.

For part (1), assume $E \vdash P : U$. This must have been derived from (Proc Input), with $E, n_1{:}W_1, ..., n_k{:}W_k \vdash P' : U$, where $U = W_1{\times}...{\times}W_k$. Moreover, for all $i \in 1..k$, if $W_i = Cap^1[T_i]$ then n_i *occurs* $P' \leq 1$. By induction hypothesis, $E, n_1{:}W_1, ..., n_k{:}W_k \vdash Q' : U$. By Lemma 8-4, $P' \equiv Q'$ implies that n_i *occurs* $P' = n_i$ *occurs* Q' for each $i \in 1..k$. Therefore, for all $i \in 1..k$, if $W_i = Cap^1[T_i]$ then n_i *occurs* $Q' \leq 1$. By (Proc Input), $E \vdash (n_1{:}W_1, ..., n_k{:}W_k).Q' : U$.

Part (2) follows by symmetric considerations.

Other cases. The other cases are almost exactly as before.

□

8-16 Proof of Proposition 5-1 (Subject Reduction)

If $E \vdash P : U$ and $P \longrightarrow Q$ then $E \vdash Q : U$.

Proof

By induction on the derivation of $E \vdash P : U$.

(Red Comm) Then $P = (n_1{:}W_1, ..., n_k{:}W_k).P' \mid \langle M_1, ..., M_k\rangle$ and $Q = P'\{n_1{\leftarrow}M_1, ..., n_k{\leftarrow}M_k\}$. Assume $E \vdash P : U$. This must have been derived from (Proc Par) with $E \vdash (n_1{:}W_1, ..., n_k{:}W_k).P' : U$ and $E \vdash \langle M_1, ..., M_k\rangle : U$. The judgment $E \vdash (n_1{:}W_1, ..., n_k{:}W_k).P' : U$ must have been derived from (Proc Input) with $E, n_1{:}W_1, ..., n_k{:}W_k \vdash P' : U$, $U = W_1{\times}...{\times}W_k$ for some $U = W_1{\times}...{\times}W_k$, and for all $i \in 1..k$, if $W_i = Cap^1[T_i]$ then n_i *occurs* $P' \leq 1$. The judgment $E \vdash \langle M_1, ..., M_k\rangle : U$ must have been derived from (Proc Output) with $E \vdash M_i : W_i'$ for each $i \in 1..k$, for some $W_1' ... W_k'$, and $U = W_1'{\times}...{\times}W_k'$. Hence, $W_i' = W_i$ for each $i \in 1..k$. By k applications of Lemma 8-14, we get $E \vdash Q : U$.

Other cases. The other cases are almost exactly as before.

□

References

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

[2] Boudol, G., **Asynchrony and the π-calculus.** *Technical Report 1702, INRIA, Sophia-Antipolis,* 1992.

[3] Cardelli, L., **Abstractions for Mobile Computation**. 1998. To appear. (See www.luca.demon.co.uk.)

[4] Cardelli, L. and A.D. Gordon, **Mobile Ambients**. In *Foundations of Software Science and Computational Structures, Maurice Nivat (Ed.),* LNCS 1378, 140-155, Springer. 1998.

[5] De Nicola, R., G. Ferrari, M. Pugliese, **Coordinating Mobile Agents via Blackboards and Access Rights**. *COORDINATION'97,* LNCS 1282, 220-237, Springer. 1997.

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

[7] Kobayashi, N., B.C. Pierce, and D.N. Turner, **Linearity and the Pi-Calculus**. *Proc ACM POPL'96,* 358-371. 1996.

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

[9] Odersky, M., **Polarized Name Passing**. *Proc FST&TCS,* Springer. 1995.

[10] Pierce, B., and D. Sangiorgi, **Typing and Subtyping for Mobile Processes**. *Mathematical Structures in Computer Science,* **6**(5), 409-454. 1996.

[11] Riely, J. and M. Hennessy, **A typed language for distributed mobile processes.** In *Proc ACM POPL'98,* 378-390. 1998.

[12] Sewell, P., **Global/Local Subtyping and Capability Inference for a Distributed π-calculus**. In *Proc ICALP'98,* Springer. 1998.

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