

# **Semantics and Types for Mobile Ambients**

Andy Gordon, Microsoft Research

(based on joint work with Luca Cardelli, Microsoft Research)

MathFit Instructional Meeting on Recent Advances in  
Semantics and Types for Concurrency: Theory & Practice  
Imperial College, July 7–9, 1998

## Goals of this Lecture

To explain:

- Background and motivations for the ambient calculus
- Programming in the untyped ambient calculus
- Equational reasoning about ambients
- Ambient calculi supporting typeful mobile computation

# Background, Motivations

## Where We Were

In the 1960s and 1970s, features such as these were explained by reduction to the  $\lambda$ -calculus:

command    variable    iteration    subroutine  
expression    value  
procedure    scope    recursion  
pointer    exception    type    polymorphism

Some outcomes:

- Impact on the design of higher-order languages like Scheme, ML and Haskell
- $\lambda$ -calculi like PCF and FPC spurred substantial mathematical developments

## Against Interpretation

Some features are hard to explain in terms of  $\lambda$ -calculi.

An alternative is to use calculi of particular programming features:

- The  $\pi$ -calculus has **process** and **channel** primitives.

Example:  $(\nu c)(\bar{c}\langle m \rangle \mid \bar{c}\langle n \rangle \mid c(x).P(x))$

The  $\pi$ -calculus has already had an impact on the design of concurrent languages.

- The object calculus has **object** and **method** primitives.

Example:  $[val = 42, inc = \zeta(s)(s.val := s.val + 1)]$

- The spi calculus has **encryption** and **decryption** primitives.

Example:  $(\nu K)(\bar{c}\langle \{M\}_K \rangle \mid c(x).case\ x\ of\ \{y\}_K\ in\ P(y))$

## Where We're Going Today

As hardware and software gain mobility, and security risks loom large, features such as these are entering APIs and programming languages:

applet    servlet    remote execution  
mobile agent    security zone    firewall  
capability    access control  
component    dynamic linking

Since existing calculi explain these features only indirectly, we develop a pure calculus of secure mobile computations: the ambient calculus.

## Ambients

- An ambient has a name
- An ambient is a bundle of active computations and passive data
- An ambient has a definite boundary, with an inside and an outside
- An ambient may be nested inside another to form a hierarchy
- An ambient moves as a whole

## Formalising Ambients

Our starting point, Milner, Parrow, and Walker's  $\pi$ -calculus:

- groups processes in a **single, contiguous, centralised** collection
- enables interaction by **shared names**, used as communication channels
- has no direct account of access control

Our ambient calculus:

- groups processes in **multiple, disjoint, distributed** ambients
- enables interaction by **shared position**, with no action at a distance
- uses **capabilities**, derived from ambient names, for access control

## Related Work

### Systems supporting mobile computations:

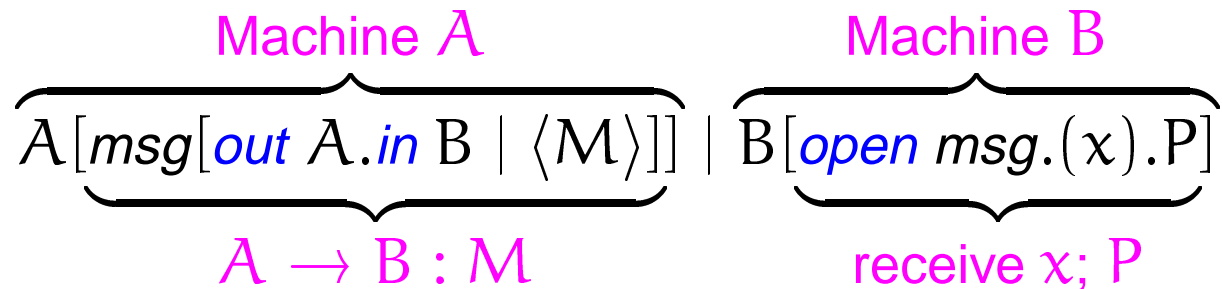
- Telescript (General Magic): while running, agents migrate from place to place
- Obliq (Cardelli): while running, objects migrate around an intranet
- Java (Sun): before running, applets migrate down a web connection

### Formalisms describing mobile computations:

- Various distributed  $\pi$ -calculi and  $\lambda$ -calculi (Amadio & Prasad; Fournet & Gonthier; Hennessy & Riely; Sewell; . . . ): the emphasis on capabilities and local mobility characterises the ambient calculus

# The Untyped Ambient Calculus

## Mobile Ambients: a packet from $A$ to $B$



- Ambients may model both machines and packets
- Ambients are mobile:  $\text{msg}[\cdot \cdot \cdot]$  moves out of  $A$  and into  $B$
- Ambients are boundaries: passage is regulated by **capabilities**

You need capability *out*  $A$  to exit  $A$ ; you need capability *in*  $B$  to enter  $B$

## Ambient Behaviour, By Example

There are four basic reduction rules in the calculus:

$$\begin{aligned}
 & A[msg[out A.in B \mid \langle M \rangle]] \mid B[open\ msg.(x).P] \\
 & \rightarrow A[] \mid msg[in B \mid \langle M \rangle] \mid B[open\ msg.(x).P] \\
 & \rightarrow A[] \mid B[msg[\langle M \rangle] \mid open\ msg.(x).P] \\
 & \rightarrow A[] \mid B[\langle M \rangle \mid (x).P] \\
 & \rightarrow A[] \mid B[P\{x \leftarrow M\}]
 \end{aligned}$$

### Mobility and Communication Primitives

$M ::=$	expression
$n$	ambient name
$in M$	can enter into $M$
$out M$	can exit out of $M$
$open M$	can open $M$
$P, Q, R ::=$	process
$(\nu n)P$	restriction
$0$	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	action
$(n_1, \dots, n_k).P$	input action
$\langle M_1, \dots, M_k \rangle$	async output action

## Subjective versus Objective Moves

We base ambient mobility on **subjective** moves:

$$\begin{aligned} n[in\ m.P \mid Q] \mid m[R] &\rightarrow m[n[P \mid Q] \mid R] \\ m[n[out\ m.P \mid Q] \mid R] &\rightarrow n[P \mid Q] \mid m[R] \end{aligned}$$

Instead, we might have adopted primitives for **objective** moves:

$$\begin{aligned} mv\ in\ n.P \mid n[Q] &\rightarrow n[P \mid Q] \\ n[mv\ out\ n.P \mid Q] &\rightarrow P \mid n[Q] \end{aligned}$$

But objective moves only move still ambients, and they allow kidnap:

$$m[P] \mid (\nu k)(k[] \mid mv\ in\ m.in\ k) \rightarrow^* (\nu k)k[m[P]]$$

## Objective Ambient Moves

The special case of objective movement of an ambient is safe, convenient, and derivable from subjective movement:

$$\textit{move } M.n[P] \stackrel{\Delta}{=} (\nu k)k[M.n[\textit{out } k.P]] \quad \text{for } k \text{ not free in } M.P$$

For example:

$$\begin{aligned} \textit{move in } m.n[P] \mid m[R] &\rightarrow (\nu k)m[k[n[\textit{out } k.P]] \mid R] \\ &\rightarrow (\nu k)m[k[] \mid n[P] \mid R] \\ &\simeq m[n[P] \mid R] \end{aligned}$$

The relation  $\simeq$  is a semantic equivalence used for garbage collection.

## Example: Encoding Channels

A communication channel may be encoded by an ambient:

$$(\nu^\pi n).P \triangleq (\nu n)(n[!\mathit{open} n] \mid P)$$

$$n\langle m \rangle \triangleq \mathit{move\ in} n.n[\langle m \rangle]$$

$$n(x).P \triangleq (\nu p)(\mathit{open} p \mid \mathit{move\ in} n.n[(x).\mathit{move\ out} n.p[P]])$$

Hence, there is an encoding of the untyped, asynchronous  $\pi$ -calculus.

# Semantics for Ambients

## A May-Testing Equivalence

Process  $P$  exhibits a top-level ambient  $n$ :  $P \downarrow n$  iff

$$P \equiv (\nu \vec{m})(P_1 \mid n[P_2]) \ \& \ n \notin \{\vec{m}\}$$

Process  $P$  may eventually exhibit  $n$ :  $P \Downarrow n$  iff

$$P \rightarrow^* Q \ \& \ Q \downarrow n$$

No process context distinguishes processes  $P$  and  $Q$ :  $P \simeq Q$  iff

$$\forall n, \text{ contexts } \mathcal{C}(\mathcal{C}[P] \Downarrow n \Leftrightarrow \mathcal{C}[Q] \Downarrow n)$$

## Examples and Non-Examples

- $p[] \not\approx q[]$

If  $\mathcal{C}\{-\} = -$  then  $\mathcal{C}\{p[]\} \Downarrow p$  but not  $\mathcal{C}\{q[]\} \Downarrow p$ .

- $\textit{open } p \not\approx \textit{open } q$

If  $\mathcal{C}\{-\} = (\nu p)(p[q[]] \mid -)$  then  $\mathcal{C}\{\textit{open } p\} \Downarrow q$  but not  $\mathcal{C}\{\textit{open } q\} \Downarrow q$ .

- $(\nu n)n[P] \simeq \mathbf{0}$  if  $n$  not free in  $P$
- $(\nu n)(n[\textit{open } n.P] \mid \textit{open } n.n[]) \simeq P$
- $(\nu n)(n[P] \mid \textit{open } n) \simeq P$  if  $n$  not free in  $P$ ?

## Example: Piloting an Agent Across a Firewall

Pre-arranged passwords  $k$ ,  $k'$ ,  $k''$  allow an agent to cross a firewall:

$$\mathit{Firewall} \triangleq (\nu w)w[k[\mathit{out} w.\mathit{in} k'.\mathit{in} w] \mid \mathit{open} k'.\mathit{open} k''.P]$$

$$\mathit{Agent} \triangleq k'[\mathit{open} k.k''[C]]$$

Assuming  $k$ ,  $k'$ ,  $k''$  do not occur in  $C$  or  $P$ , and  $w$  does not occur in  $C$ , we get the safety property:

$$(\nu k k' k'')(\mathit{Agent} \mid \mathit{Firewall}) \simeq (\nu w)w[C \mid P]$$

# Types for Ambients

## Orientation: Types

The purpose of a type system is to prevent execution errors during the running of well-typed programs.

Typed languages emerged in the 1960s and 70s: Pascal, Algol 68, Simula, ML. Mostly, typing in these languages prevents **accidental** execution errors, e.g.,  $1.0 + \text{"fred"}$ .

Recently, Java has popularised typing for mobile code. As well as preventing accidents, typing in Java prevents **malicious** execution errors, e.g., formatting the C drive.

## Related Work

Our work borrows ideas from previous treatments of types for the  $\lambda$ -calculus and the  $\pi$ -calculus.

In particular, Milner's sorts for  $\pi$ , Pierce and Sangiorgi's type system for  $\pi$ , and Kobayashi, Pierce, and Turner's linear type system for  $\pi$ .

Some quite sophisticated type systems are being investigated for mobile computation, e.g., by Sewell, Hennessy and Riely, Jeffrey, . . .

Our approach is to investigate several simple systems that regulate exchanges, mobility, security levels, etc., and attempt to integrate them into a coherent whole.

## Motivation for Exchange Types

In the untyped calculus, certain processes arise that make no sense:

- Process  $in\ n[P]$  uses a capability as an ambient name
- Process  $n.P \mid n[Q]$  uses an ambient name as a capability

In an implementation, these processes are execution errors.

To avoid these errors, we regulate the types of messages a process may **exchange**, that is, input or output.

## Typing Input and Output

If a message  $M$  has message type  $W$ , then  $\langle M \rangle$  is a process that exchanges  $W$  messages.

If  $M : W$  then  $\langle M \rangle : W$ .

If  $P$  is a process that exchanges  $W$  messages, then  $(x:W).P$  is also a process that exchanges  $W$  messages.

If  $P : W$  then  $(x:W).P : W$ .

## Typing Parallelism

Process  $\mathbf{0}$  exchanges messages of any type, since it exchanges none.

$\mathbf{0} : T$  for all  $T$ .

If  $P$  and  $Q$  are processes that exchange  $T$  messages, so is  $P \mid Q$ .

If  $P : T$  and  $Q : T$  then  $P \mid Q : T$ .

If  $P : T$  then  $!P : T$ .

These rules ensure matching of the types of inputs and outputs from processes running in parallel.

## Typing Ambients

An expression of type  $Amb[T]$  names an ambient inside which  $T$  messages are exchanged.

If  $M$  is such an expression, and  $P$  is a process that exchanges  $T$  messages, then  $M[P]$  is correctly typed.

If  $M : Amb[T]$  and  $P : T$  then  $M[P] : S$  for all  $S$ .

An ambient exchanges no messages, so it may be assigned any type.

## Typing Capabilities

An expression of type  $Cap[T]$  is a capability that may unleash exchanges of type  $T$ .

If  $M : Cap[T]$  and  $P : T$  then  $M.P : T$ .

If ambients named  $n$  exchange  $T$  messages, then the capability  $open\ n$  may unleash these exchanges.

If  $n : Amb[T]$  then  $open\ n : Cap[T]$ .

Capabilities  $in\ n$  and  $out\ n$  unleash no exchanges.

If  $n : Amb[S]$  then  $in\ n : Cap[T]$  for all  $T$ .

If  $n : Amb[S]$  then  $out\ n : Cap[T]$  for all  $T$ .

## Exchange Types

### Types:

$W ::=$	message types
$Amb[T]$	ambient name allowing $T$ exchanges
$Cap[T]$	capability unleashing $T$ exchanges
$S, T ::=$	exchange types
$Shh$	no exchange
$W_1 \times \dots \times W_k$	tuple exchange

- A quiet ambient,  $Amb[Shh]$ , and a harmless capability,  $Cap[Shh]$
- An ambient allowing exchange of harmless capabilities:  $Amb[Cap[Shh]]$
- A capability unleashing exchanges of names of quiet ambients:  $Cap[Amb[Shh]]$

## Properties of Exchange Types

**Proposition** (Soundness) If  $P : T$  and  $P \rightarrow Q$  then  $Q : T$ .

Hence, execution errors like *in*  $n[P]$  and  $n.P \mid n[Q]$  cannot arise during a computation, since they are not typeable.

## Examples of Typing

Packet from  $\bar{A}$  to  $B$ :

If  $\bar{A} : \mathit{Amb}[\mathit{Shh}]$ ,  $B$ ,  $\mathit{msg} : \mathit{Amb}[\mathit{Msg}]$ , and  $M, P : \mathit{Msg}$  then  
 $\bar{A}[\mathit{msg}[\underbrace{\mathit{out} \bar{A}.\mathit{in} B}_{\mathit{Cap}[\mathit{Msg}]} \mid \langle M \rangle] \mid B[\underbrace{\mathit{open} \mathit{msg} . (x:\mathit{Msg}).P}_{\mathit{Cap}[\mathit{Msg}]}] : \mathit{Shh}.$

Objective ambient move:

If  $M : \mathit{Cap}[T]$  and  $n[P] : S$  then  $\mathit{move} M.n[P] : S.$

## Encoding Simple Typed $\pi$ -Calculus in Ambients

Translated types  $\llbracket W \rrbracket$ , translated processes  $\llbracket P \rrbracket$ :

$$\llbracket Ch[W_1, \dots, W_k] \rrbracket \triangleq Amb[\llbracket W_1 \rrbracket \times \dots \times \llbracket W_k \rrbracket]$$

$$\llbracket (\nu^\pi n:W)P \rrbracket \triangleq (\nu n:\llbracket W \rrbracket)(n[!open\ n] \mid \llbracket P \rrbracket)$$

$$\llbracket n\langle n_1, \dots, n_k \rangle \rrbracket \triangleq move\ in\ n.n[\langle n_1, \dots, n_k \rangle]$$

$$\llbracket n(n_1, \dots, n_k).P \rrbracket \triangleq (\nu p:Amb[Shh])(open\ p \mid \\ move\ in\ n.n[(n_1, \dots, n_k).move\ out\ n.p[\llbracket P \rrbracket]])$$

$$\llbracket P \mid Q \rrbracket \triangleq \llbracket P \rrbracket \mid \llbracket Q \rrbracket$$

$$\llbracket !P \rrbracket \triangleq !\llbracket P \rrbracket$$

**Proposition (Soundness)** If  $P$  well-typed then  $\llbracket P \rrbracket : Shh$ .

## Expressiveness of Exchange Types

Some encodings supported by exchange types:

- A typed form of Milner's  $\pi$ -calculus:  $Ch[W] \stackrel{\Delta}{=} Amb[W]$
- The simple typed  $\lambda$ -calculus:  $A \rightarrow B \stackrel{\Delta}{=} Ch[A, Ch[B]]$
- A typed fragment of Telescript, a language of mobile agents

**Exercise:** Is the typed ambient calculus Turing complete?

# Linear Exchange Types

## Motivation: An Online Postbox

Consider a server allowing agents to send real letters.

Virtual postage stamps are derived from  $queen : \mathit{Amb}[Letter]$ , a secret, and are provided along the channel  $init : \mathit{Chan}[Stamp]$

$$postbox \triangleq !queen[(letter:Letter).deliver\ letter]$$

$$Stamp \triangleq Cap[Letter]$$

$$stamp : Stamp \triangleq open\ queen$$

$$alice : Letter \triangleq init(\chi:Stamp).\chi.\langle \text{“Dear Bob...”} \rangle$$

The server posts a letter for the correspondent:

$$postbox \mid alice \mid init\langle stamp \rangle \rightarrow^* postbox \mid deliver\ \text{“Dear Bob...”}$$

## Abusing Virtual Postage Stamps

A duplicitous correspondent:

$$mallory \stackrel{\Delta}{=} init(x).x.x.(\langle \text{“Dear Alice...”} \rangle \mid \langle \text{“Dear Bob...”} \rangle)$$

Sending two letters with the one stamp:

$$postbox \mid mallory \mid init \langle stamp \rangle$$

$$\rightarrow^* postbox \mid stamp.stamp.(\langle \text{“Dear Alice...”} \rangle \mid \langle \text{“Dear Bob...”} \rangle)$$

$$\rightarrow^* postbox \mid deliver \text{“Dear Alice...”} \mid deliver \text{“Dear Bob...”}$$

## Virtual Postage Stamps have Linear Type

Like other systems based on capabilities, the untyped ambient calculus provides no bound on the number of uses of a capability.

Runtime usage checks could be expensive.

Instead, inspired by ideas from Girard's linear logic, we propose a type system to verify correct usage of stamps and other capabilities.

Our system can be used to verify untrusted applets before they are run on our online postbox server: it accepts *alice* but rejects *mallory*.

## Linear Capabilities, Unlimited Ambient Names

Our system regulates the following, rather simple principle:

- An input of type  $Cap[T]$  may be exercised at most once
- An input of type  $Amb[T]$  may be exercised as often as desired

For example:

- Disallowed:  $(x:Cap[T]).(\langle x \rangle \mid \langle x \rangle)$ ,  $(x:Cap[T]).(\langle x \rangle \mid n[x.P])$
- Allowed:  $(x:Amb[T]).(x[P] \mid x[Q])$ ,  $(x:Amb[T]).n[in\ x.in\ x.P]$

## Linearising the Type System

We track whether a name occurs never, once, or many times in a process. For example:

$$n \text{ occurs } (m[] \mid (\nu n)n[]) = 0$$

$$n \text{ occurs } m[n.\mathbf{0}] = 1$$

$$n \text{ occurs } (m[n.\mathbf{0}] \mid \langle n \rangle) = \omega$$

Each input capability can be used no more than once:

If  $E, n:\mathit{Cap}[T] \vdash P : \mathit{Cap}[T]$  and  $n \text{ occurs } P \leq 1$   
 then  $E \vdash (n:\mathit{Cap}[T]).P : \mathit{Cap}[T]$ .

That's all!

## Properties of Linear Exchange Types

**Proposition** (Soundness) If  $P : T$  and  $P \rightarrow Q$  then  $Q : T$ .

Hence, execution errors like *mallory* cannot arise during a computation, since they are not typeable.

## Summary

A goal of developing our calculus is to prototype a flexible, precise, secure, and typeful programming model for mobile software components.

A semantic equivalence allows statement and proof of security properties.

Type systems regulate input/output, use of capabilities, and mobility, preventing both accidental and malicious errors.