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
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
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 \textit{process} and \textit{channel} primitives.
  
  Example: \((\nu c)(\overline{c}\langle m \rangle | \overline{c}\langle n \rangle | c(x).P(x))\)

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

- The object calculus has \textit{object} and \textit{method} primitives.
  
  Example: \([val = 42, inc = \zeta(s)(s.val := s.val + 1)]\)

- The spi calculus has \textit{encryption} and \textit{decryption} primitives.
  
  Example: \((\nu K)(\overline{c}\langle\{M\}_K \rangle | c(x).case x of \{y\}_K in P(y))\)
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.
• 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 $B$

Machine $A$

$A[\text{msg[out A.in B | } \langle M \rangle\text{]}] | B[\text{open msg.(x).P}]$

$A \rightarrow B : M$

Machine $B$

receive $x; P$

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

You need capability $\text{out } A$ to exit $A$; you need capability $\text{in } B$ to enter $B$
There are four basic reduction rules in the calculus:

\[
A \left[ \text{msg[out} A\text{.in B}} \mid \langle M \rangle \right] \mid B[\text{open msg.(x).P}]
\]

\[
\rightarrow A \parallel \mid \text{msg[in} B \mid \langle M \rangle \mid B[\text{open msg.(x).P}]
\]

\[
\rightarrow A \parallel \mid B[\text{msg[} \langle M \rangle \mid \text{open msg.(x).P}]
\]

\[
\rightarrow A \parallel \mid B[\langle M \rangle \mid (x).P]
\]

\[
\rightarrow A \parallel \mid B[P\{x \leftarrow M\}]
\]
### Mobility and Communication Primitives

\[ M ::= \]

- \( n \)                     
  ambient name
- \( in\ M \)                 
  can enter into \( M \)
- \( out\ M \)                
  can exit out of \( M \)
- \( open\ M \)               
  can open \( M \)

\[ P, Q, R ::= \]

- \( (\forall n)P \)          
  restriction
- \( 0 \)                     
  inactivity
- \( P \mid Q \)              
  composition
- \( !P \)                    
  replication
- \( M[P] \)                  
  ambient
- \( M.P \)                   
  action
- \( (n_1, \ldots, n_k).P \)  
  input action
- \( \langle M_1, \ldots, M_k \rangle \)  
  async output action
Subjective versus Objective Moves

We base ambient mobility on subjective moves:

\begin{align*}
\text{n[in } m.P \mid Q \mid m[R] & \rightarrow m[n[P \mid Q] \mid R] \\
m[n[\text{out } m.P \mid Q] \mid R] & \rightarrow n[P \mid Q] \mid m[R]
\end{align*}

Instead, we might have adopted primitives for objective moves:

\begin{align*}
\text{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{align*}

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

\begin{align*}
m[P] \mid (\forall k)(k[] \mid \text{mv in } m.in k) & \rightarrow^* (\forall k)k[m[P]]
\end{align*}
Objective Ambient Moves

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

\[
move \ M.n[P] \overset{\triangle}{=} (\forall k) k[M.n[\text{out} \ k.P]] \quad \text{for } k \text{ not free in } M.P
\]

For example:

\[
move \ in \ m.n[P] \mid m[R] \rightarrow (\forall k)m[k[n[\text{out} \ k.P]] \mid R]
\]

\[
\rightarrow (\forall k)m[k] \mid n[P] \mid R
\]

\[
\simeq m[n[P] \mid R]
\]

The relation \(\simeq\) is a semantic equivalence used for garbage collection.
Example: Encoding Channels

A communication channel may be encoded by an ambient:

\[
(\forall \pi n) P \triangleq (\forall n)(n[!\text{open } n] \mid P)
\]

\[
n(m) \triangleq \text{move in } n. n[\langle m \rangle]
\]

\[
n(x). P \triangleq (\forall p)(\text{open } p \mid \text{move in } n. n[\langle x \rangle. \text{move out } n.p[P]])
\]

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

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

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

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

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

$$\forall n, \text{contexts } C(C[P] \downarrow n \Leftrightarrow C[Q] \Downarrow n)$$
Examples and Non-Examples

- \( p [] \not\equiv q [] \)
  
  If \( \mathcal{C}\{\_\} = \_ \) then \( \mathcal{C}\{p []\} \downarrow p \) but not \( \mathcal{C}\{q []\} \downarrow p \).

- \( \text{open } p \not\equiv \text{open } q \)
  
  If \( \mathcal{C}\{\_\} = (\forall p)(p[q []] \mid \_) \) then \( \mathcal{C}\{\text{open } p\} \downarrow q \) but not \( \mathcal{C}\{\text{open } q\} \downarrow q \).

- \( (\forall n)n[P] \simeq 0 \) if \( n \) not free in \( P \)

- \( (\forall n)(n[\text{open } n.P] \mid \text{open } n.n []) \simeq P \)

- \( (\forall n)(n[P] \mid \text{open } n) \simeq P \) 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:

\[
\text{Firewall} \triangleq (\forall w) w[k[\text{out } w.\text{in } k'.\text{in } w] \mid \text{open } k'.\text{open } k''.P]
\]

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

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

\[
(\forall k\ k'\ k'') (\text{Agent} \mid \text{Firewall}) \simeq (\forall w) w[C \mid P]
\]
Types for Ambients
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 + “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.
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 Sangioergi’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 $\text{in } \pi[P]$ uses a capability as an ambient name
- Process $\pi.P | \pi[Q]$ 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 0 exchanges messages of any type, since it exchanges none.

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

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

If $P : T$ and $Q : T$ then $P | Q : T$.  
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.


An ambient exchanges no messages, so it may be assigned any type.
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 $\eta$ exchange $T$ messages, then the capability $open \ \eta$ may unleash these exchanges.

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

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

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

If $\eta : Amb[S]$ then $out \ \eta : Cap[T]$ 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

\[ W_1 \times \cdots \times W_k \]
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 $\textit{in} \; n[P]$ and $n.P \mid n[Q]$ cannot arise during a computation, since they are not typeable.
Examples of Typing

Packet from \( A \) to \( B \):

If \( A : Amb[Shh] \), \( B \), \( msg : Amb[Msg] \), and \( M \), \( P : Msg \) then

\[
A [msg[out \: A \: in \: B] \mid \langle M \rangle] \mid B [open \: msg \: . (x:Msg) \cdot P] : Shh.
\]

Objective ambient move:

If \( M : Cap[T] \) and \( n[P] : S \) then \( move \: M.n[P] : S \).
Encoding Simple Typed $\pi$-Calculus in Ambients

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

$$\llbracket Ch[W_1, \ldots, W_k] \rrbracket \overset{\Delta}{=} Amb[\llbracket W_1 \rrbracket \times \cdots \times \llbracket W_k \rrbracket]$$

$$\llbracket (\forall \pi \ n : W) P \rrbracket \overset{\Delta}{=} (\forall n : [W])(n[open n] \mid [P])$$

$$\llbracket n(n_1, \ldots, n_k) \rrbracket \overset{\Delta}{=} move\ in\ n.n[\langle n_1, \ldots, n_k \rangle]$$

$$\llbracket n(n_1, \ldots, n_k).P \rrbracket \overset{\Delta}{=} (\forall p : Amb[Shh])(open p \mid$$

$$move\ in\ n.n[\langle n_1, \ldots, n_k \rangle].move\ out\ n.p[[P]])$$

$$\llbracket P \mid Q \rrbracket \overset{\Delta}{=} [P] \mid [Q]$$

$$\llbracket ![P] \rrbracket \overset{\Delta}{=} ![P]$$

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: $Ch[\mathcal{W}] \triangleq Amb[\mathcal{W}]$

- The simple typed $\lambda$-calculus: $A \rightarrow B \triangleq Ch[A, Ch[B]]$

- 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 $queen : Amb[Letter]$, a secret, and are provided along the channel $init : Chan[Stamp]$

\[
\text{postbox} \triangleq !queen[(\text{letter}:Letter).\text{deliver letter}]
\]

\[
\text{Stamp} \triangleq Cap[Letter]
\]

\[
\text{stamp} : \text{Stamp} \triangleq \text{open queen}
\]

\[
\text{alice} : \text{Letter} \triangleq \text{init}(x:\text{Stamp}).x.(\text{“Dear Bob...”})
\]

The server posts a letter for the correspondent:

\[
\text{postbox} \mid \text{alice} \mid \text{init}(\langle \text{stamp} \rangle) \rightarrow^* \text{postbox} \mid \text{deliver “Dear Bob...”}
\]
Abusing Virtual Postage Stamps

A duplicitous correspondent:

\[ \text{mallory} \overset{\Delta}{=} \text{init}(x).x.x.(\langle \text{"Dear Alice..."} \rangle \mid \langle \text{"Dear Bob..."} \rangle) \]

Sending two letters with the one stamp:

\[
\begin{align*}
\text{postbox} \mid \text{mallory} \mid \text{init}(\text{stamp}) \\
\to^* \text{postbox} \mid \text{stamp}.\text{stamp.}(\langle \text{"Dear Alice..."} \rangle \mid \langle \text{"Dear Bob..."} \rangle) \\
\to^* \text{postbox} \mid \text{deliver} \text{"Dear Alice..."} \mid \text{deliver} \text{"Dear Bob..."}
\end{align*}
\]
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 $\text{Cap}[T]$ may be exercised at most once
- An input of type $\text{Amb}[T]$ may be exercised as often as desired

For example:

- Disallowed: $(x:Cap[T]).(\langle x \rangle | \langle x \rangle)$, $(x:Cap[T]).(\langle x \rangle | n[x.P])$
- Allowed: $(x:Amb[T]).(x[P] | x[Q])$, $(x:Amb[T]).n[in\ x.in\ x.P]$
Linearising the Type System

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

\[
\begin{align*}
\text{n occurs } & \ (m[] \mid (\forall n)n[]) = 0 \\
\text{n occurs } & \ m[n.0] = 1 \\
\text{n occurs } & \ (m[n.0] \mid \langle n \rangle) = \omega
\end{align*}
\]

Each input capability can be used no more than once:

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

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.