Abstractions for Mobile Computation

Luca Cardelli

with Andrew D. Gordon

Microsoft Research

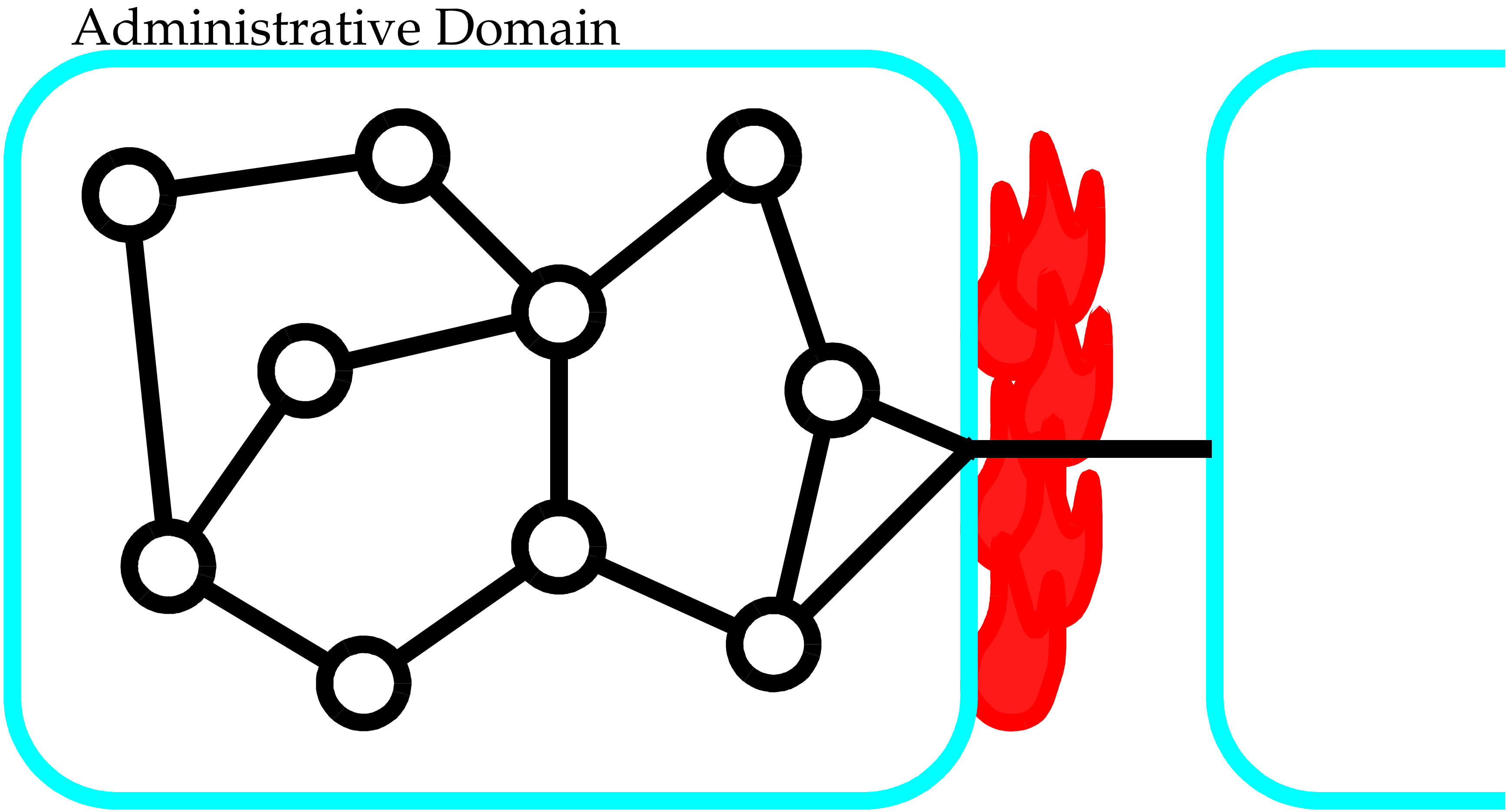
Outline

- Understanding Mobility
  - ~ Virtual mobility
  - ~ Physical mobility
  - ~ Both together over wide areas
- Modeling mobility
  - ~ Why previous formalisms are not good enough
  - ~ The ambient calculus
- Applications and future directions
  - ~ Study of combined security and mobility properties
  - ~ Libraries and languages for wide-area networks

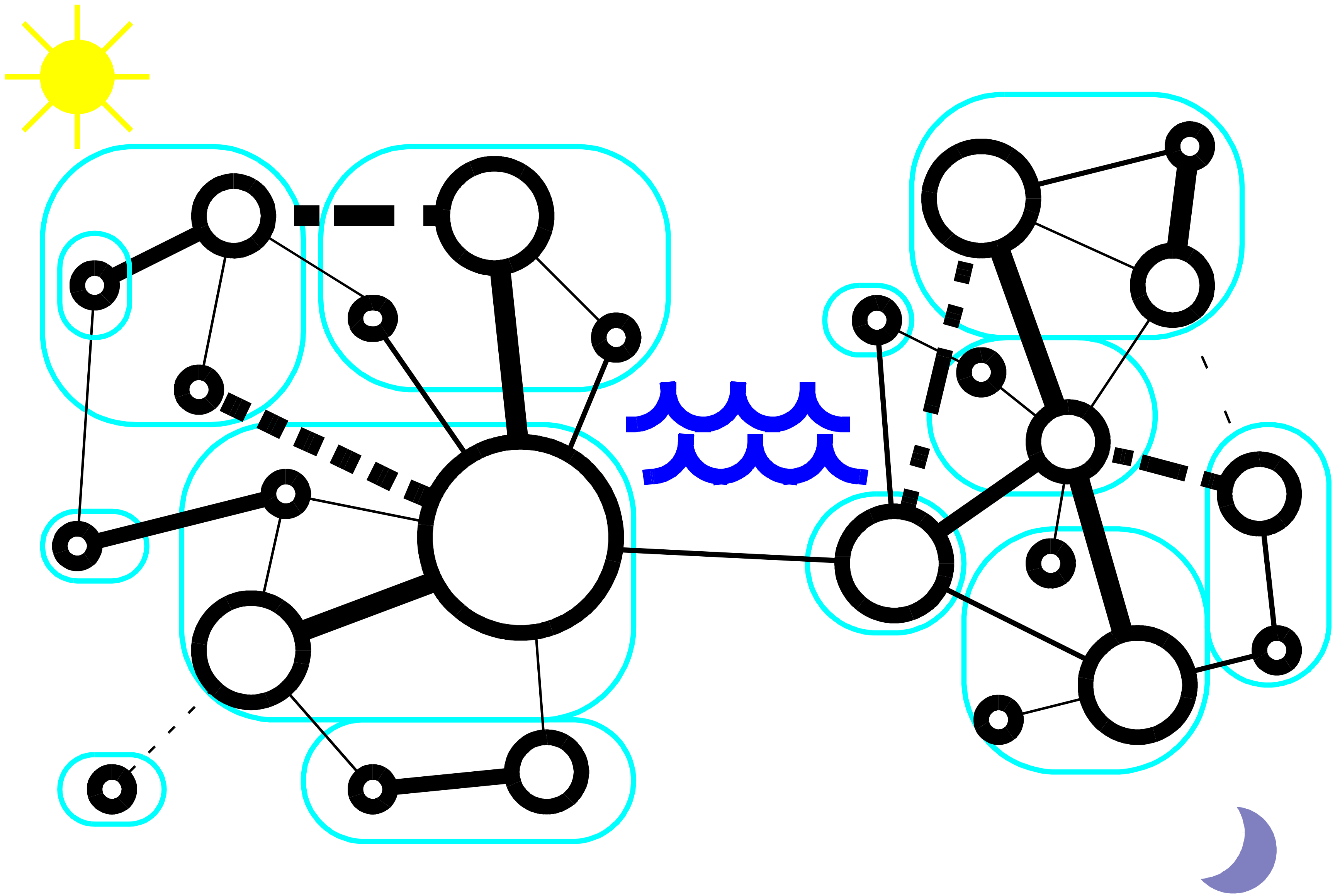
Three Mental Pictures

- Local area networks
- Wide area networks
- Mobile networks

LANs and (Traditional) Distributed Computing



The Web



WAN Characteristics

- Internet/Web: a federated WAN infrastructure that spans the planet. We would like to program it.
- Unfortunately, federated WANs violate many familiar assumptions about the behavior of distributed systems.
- Three phenomena that remain largely hidden in LANs become readily observable:
  - ~ ***Virtual locations***.
  - ~ ***Physical locations***.
  - ~ ***Bandwidth fluctuations***.
- Another phenomenon becomes unobservable:
  - ~ ***Failures***.

A WAN is not a big LAN

- To emulate a LAN on top of a WAN we would have to:
  - ~ ***(A) Hide virtual locations***. By semi-transparent security. But is it possible to guarantee the integrity of mobile code?
  - ~ ***(B) Hide physical locations.*** Cannot "hide" the speed of light, other than by slowing down the whole network.
  - ~ ***(C) Hide bandwidth fluctuations***. Service guarantees eliminate bandwidth fluctuations, but introduce access failures.
  - ~ ***(D) Reveal failures.*** Impossible in principle, since the Web is an asynchronous network.
- In summary: (A) may be unsolvable for mobile code; (B) is only solvable (in full) by introducing unacceptable delays; (C) can be solved in a way that reduces it to (D); (D) is unsolvable in principle, while probabilistic solutions run into point (B).

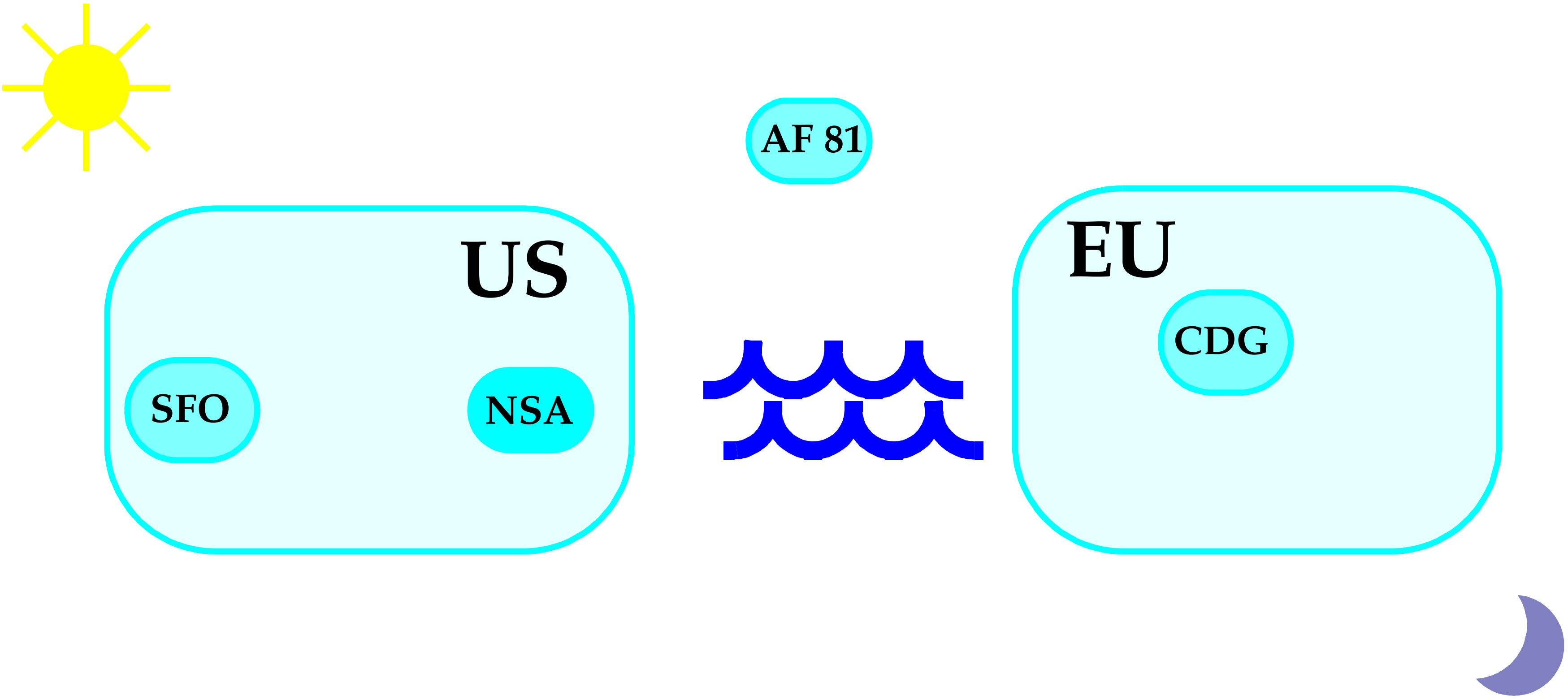
Observables

- WAN *observables* are different (and not reducible to) LAN observables.
- Observables determine programming constructs, and therefore influence programs and programming languages.
- We need a complete set of programming constructs that can detect and react to the available observables and, of course, we do not want programming constructs that attempt to detect or react to non-observables.

Mobile Computation

- Mobile computation can cope with the observables characteristic of a wide-area network such as the Web.
  - ~ ***Virtual locations***. Trust mechanisms to cross virtual barriers.
  - ~ ***Physical locations***. Mobility to optimize placement.
  - ~ ***Bandwidth fluctuations***. Mobility to split applications and establish optimized communication protocols.
  - ~ ***Failures***. Running around or away from failures.

Mobile Computing



- Mobile devices also move computations. In this sense, we cannot avoid the issues raised by mobile computation.

Mobility Postulates

- Separate locations exist. They may be difficult to reach.
- Since different locations have different properties, both people and programs (and sublocations!) will want to move between them.
- Barriers to mobility will be erected to preserve certain properties of certain locations.
- Some people and some programs will still need to cross those barriers.

This is the situation Wide-Area Languages have to cope with.

Related Work

- Broadly classifiable in two categories:
  - ~ Agents (Actors, Process Calculi, Telescript, etc.)
  - ~ Spaces (Linda, Distributed Lindas, JavaSpace, etc.)
- (With the work on Ambients, we aim to unify and extend those basic concepts.)

Some of my own work

- *Service Combinators* (with Rowan Davies)

  → *WebL* (Hannes Marais et al.)

  ~ Control constructs for handling bandwidth fluctuations from the point of view of a static client.

- *Ambients* (with Andrew D. Gordon)

  ~ Control constructs for handling mobility of entire subsystems, barrier crossing, and security.

Modeling Mobility

- It's all about barriers:
  - ~ Locality = **barrier topology**.
  - ~ Process mobility = **barrier crossing**.
  - ~ Security = **(In)ability to cross barriers**.
  - ~ Interaction by **shared position within a barrier**, with no action at a distance.

Formalisms for Concurrency/Distribution

- CSP/CCS. (Static/immutable connectivity.)
- π-calculus. (Channel mobility.)
  N.B. "mobility" in this context is not process mobility.
- Process mobility is reduced to channel mobility.

- Ambient Calculus:
  Process mobility = Barrier crossing.

... in particular, π

- In the π-calculus (our starting point):
  - ~ processes exist in a single **contiguous** location
  - ~ interaction is by **shared names**, used as I/O channels
  - ~ there is no direct account of access control

- In our ambient calculus:
  - ~ processes exist in multiple **disjoint** locations
  - ~ interaction is by **shared position**, with no action at a distance
  - ~ **capabilities**, derived from ambient names, regulate access

Formalisms for Locality

- Join calculus. (Channel mobility and locality.)
- Various calculi with failure. (Locality = Partial Failure.)

- Ambient calculus:
  Locality = Barrier topology.

Formalisms for Security

- (BAN logic, etc.)
- Spi-calculus. (Channel mobility and cryptography)

- Ambient calculus:
  Security = (In)ability to cross barriers.

Ambients

- We want to capture in an abstract way, notions of locality, of mobility, and of ability to cross barriers.
- An *ambient* is a place, delimited by a boundary, where computation happens.
- Ambients have a *name*, a collection of local *processes*, and a collection of *subambients*.
- Ambients can move in an out of other ambients, subject to *capabilities* that are associated with ambient names.
- Ambient names are unforgeable (as in π and spi).

The Ambient Calculus

$P ::=$	$(\nu n)\ P$	new name n in a scope	scoping
	0	inactivity	standard in process calculi
	$P \mid P$	parallel	
	$!P$	replication	data structures
	$M[P]$	ambient	
	$M.P$	exercise a capability	ambient-specific
	$(n).P$	input locally, bind to n	actions
	$\langle M \rangle$	output locally (async)	ambient I/O
$M ::=$	n	name	
	$in\ M$	entry capability	
	$out\ M$	exit capability	basic capabilities
	$open\ M$	open capability	
	ε	empty path	
	$M.M'$	composite path	useful with I/O

Semantics

- Behavior
  - ~ The semantics of the ambient calculus is given in non-deterministic "chemical style" (as in Berry&Boudol's Chemical Abstract Machine, and in Milner's π-calculus).
  - ~ The semantics is factored into a reduction relation $P \longrightarrow P'$ describing the evolution of a process P into a process P', and a process equivalence indicated by $Q \equiv Q'$.
  - ~ Here, $\longrightarrow$ is real computation, while $\equiv$ is "rearrangement".
- Equivalence
  - ~ On the basis of behavior, a substitutive *observational equivalence*, $P \approx Q$, is defined between processes.
  - ~ Standard process calculi reasoning techniques (context lemmas, bisimulation, etc.) can be adapted.

Parallel

- Parallel execution is denoted by a binary operator:

$$P \mid Q$$

- It is commutative and associative:

$$\begin{aligned} P \mid Q &\equiv Q \mid P \\ (P \mid Q) \mid R &\equiv P \mid (Q \mid R) \end{aligned}$$

- It obeys the reduction rule:

$$P \longrightarrow Q \;\Rightarrow\; P \mid R \longrightarrow Q \mid R$$

Replication

- Replication is a technically convenient way of representing iteration and recursion.

$$!P$$

- It denotes the unbounded replication of a process P.

$$!P \;\equiv\; P \mid !P$$

- There are no reduction rules for $!P$; in particular, the process P under ! cannot begin to reduce until it is expanded out as $P \mid !P$.

Restriction

- The restriction operator creates a new (forever unique) ambient name n within a scope P.

$$(\nu n)P$$

- As in the π-calculus, the (νn) binder can float as necessary to extend or restrict the scope of a name. E.g.:

$$(\nu n)(P \mid Q) \quad \equiv \quad P \mid (\nu n)Q \qquad \text{if } n \notin fn(P)$$

- Reduction rule:

$$P \longrightarrow Q \quad \Rightarrow \quad (\nu n)P \longrightarrow (\nu n)Q$$

Inaction

- The process that does nothing:

$$\mathbf{0}$$

- Some garbage-collection equivalences:

$$P \mid \mathbf{0} \equiv P$$
$$!\mathbf{0} \equiv \mathbf{0}$$
$$(\nu n)\mathbf{0} \equiv \mathbf{0}$$

- This process does not reduce.

Ambients

- An ambient is written as follows, where n is the name of the ambient, and P is the process running inside of it.

$$n[P]$$

- In $n[P]$, it is understood that P is actively running:

$$P \longrightarrow Q \quad \Rightarrow \quad n[P] \longrightarrow n[Q]$$

- Multiple ambients may have the same name, (e.g., replicated servers).

Actions and Capabilities

- Operations that change the hierarchical structure of ambients are sensitive. They can be interpreted as the crossing of firewalls or the decoding of ciphertexts.
- Hence these operations are restricted by *capabilities*.

$$M.\, P$$

  This executes an action regulated by the capability M, and then continues as the process P.

- The reduction rules for $M.\, P$ depend on M.

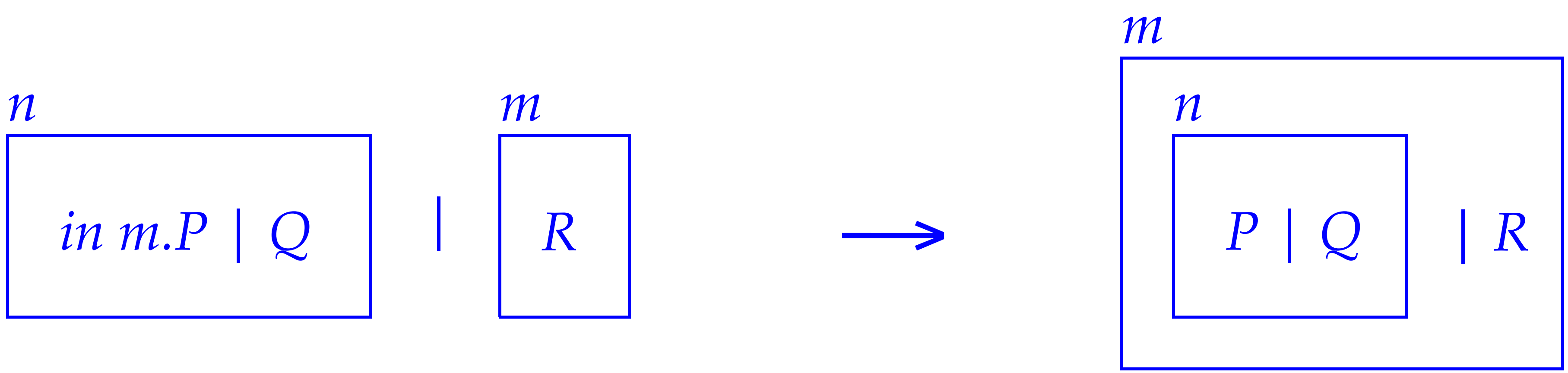
Entry Capability

- An entry capability, *in m*, can be used in the action:

$$in\ m.\ P$$

- The reduction rule (non-deterministic and blocking) is:

$$n[in\ m.\ P \mid Q] \mid m[R] \ \longrightarrow\ m[n[P \mid Q] \mid R]$$



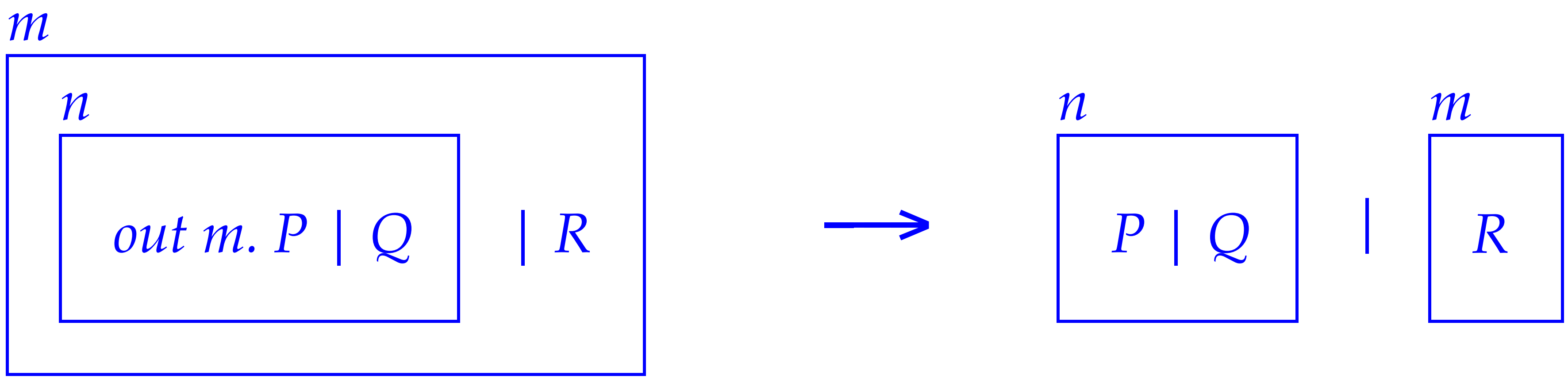
Exit Capability

- An exit capability, *out m*, can be used in the action:

$$out\ m.\ P$$

- The reduction rule (non-deterministic and blocking) is:

$$m[n[out\ m.\ P \mid Q] \mid R] \;\longrightarrow\; n[P \mid Q] \mid m[R]$$



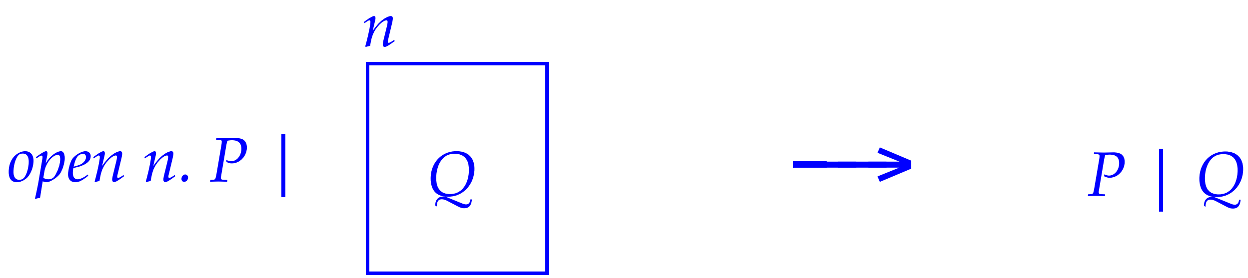
Open Capability

- An opening capability, *open m*, can be used in the action:

$$\mathit{open}\ n.\ P$$

- The reduction rule (non-deterministic and blocking) is:

$$\mathit{open}\ n.\ P \mid n[Q] \;\longrightarrow\; P \mid Q$$



- An *open* operation may be upsetting to both P and Q above.
  - ~ From the point of view of P, there is no telling in general what Q might do when unleashed.
  - ~ From the point of view of Q, its environment is being ripped open.
- Still, this operation is relatively well-behaved because:
  - ~ The dissolution is initiated by the agent $open\ n.\ P$, so that the appearance of Q at the same level as P is not totally unexpected;
  - ~ $open\ n$ is a capability that is given out by n, so $n[Q]$ cannot be dissolved if it does not wish to be.

Design Principle

- An ambient should not get killed or trapped unless:
  - ~ It talks too much. (By making its capabilities public.)
  - ~ It poisons itself. (By opening an untrusted intruder.)
  - ~ It steps into quicksand. (By entering an untrusted ambient.)
- Some natural primitives violate this principle. E.g.:

$$n[\underline{burst\ n.\ P} \mid Q] \;\longrightarrow\; P \mid Q$$

Then a mere *in* capability gives a kidnapping ability:

$$entrap(M) \;\triangleq\; (\nu\, k\, m)\,(m[M.\ burst\ m.\ in\ k] \mid k[\,])$$

$$\begin{aligned} entrap(in\ n) \mid n[P] \;&\longrightarrow^*\; (\nu k)\,(n[in\ k \mid P] \mid k[\,]) \\ &\longrightarrow^*\; (\nu k)\,k[n[P]] \end{aligned}$$

- Morale
  - ~ One can imagine lots of different mobility primitives.
  - ~ But one must think hard about the "security" implications of combinations of these primitives.

Ambient I/O

- Local anonymous communication within an ambient:

$(x).\ P$	input action
$\langle M \rangle$	async output action

- We have the reduction:

$$(x).\ P \mid \langle M \rangle \longrightarrow P\{x \leftarrow M\}$$

- This mechanism fits well with the ambient intuitions.
  - ~ Long-range communication, like long-range movement, should not happen automatically because messages may have to cross firewalls and other obstacles. (C.f., Telescript.)
  - ~ Still, this is sufficient to emulate communication over named channels, etc.

Reduction Summary

$$n[in\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R] \qquad \text{(Red In)}$$

$$m[n[out\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R] \qquad \text{(Red Out)}$$

$$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q \qquad \text{(Red Open)}$$

$$(n).\ P \mid \langle M \rangle \longrightarrow P\{n \leftarrow M\} \qquad \text{(Red Comm)}$$

$$P \longrightarrow Q \ \Rightarrow\ (\nu n)P \longrightarrow (\nu n)Q \qquad \text{(Red Res)}$$

$$P \longrightarrow Q \ \Rightarrow\ n[P] \longrightarrow n[Q] \qquad \text{(Red Amb)}$$

$$P \longrightarrow Q \ \Rightarrow\ P \mid R \longrightarrow Q \mid R \qquad \text{(Red Par)}$$

$$P' \equiv P,\ P \longrightarrow Q,\ Q \equiv Q' \ \Rightarrow\ P' \longrightarrow Q' \qquad \text{(Red} \equiv\text{)}$$

$\longrightarrow^*$ reflexive and transitive closure of $\longrightarrow$

In addition, we identify terms up to renaming of bound names:

$$(\nu n)P \ = \ (\nu m)P\{n \leftarrow m\} \quad \text{if } m \notin fn(P)$$

$$(n).P \ = \ (m).P\{n \leftarrow m\} \quad \text{if } m \notin fn(P)$$

Structural Congruence Summary

$P \equiv P$	(Struct Refl)
$P \equiv Q \quad \Rightarrow \quad Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \quad \Rightarrow \quad P \equiv R$	(Struct Trans)
$P \equiv Q \quad \Rightarrow \quad (\nu n)P \equiv (\nu n)Q$	(Struct Res)
$P \equiv Q \quad \Rightarrow \quad P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \quad \Rightarrow \quad !P \equiv !Q$	(Struct Repl)
$P \equiv Q \quad \Rightarrow \quad M[P] \equiv M[Q]$	(Struct Amb)
$P \equiv Q \quad \Rightarrow \quad M.P \equiv M.Q$	(Struct Action)
$P \equiv Q \quad \Rightarrow \quad (n).P \equiv (n).Q$	(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)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Struct Res Res)
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \quad \text{if } n \notin fn(P)$	(Struct Res Par)
$(\nu n)(m[P]) \equiv m[(\nu n)P] \quad \text{if } n \neq m$	(Struct Res Amb)
$P \mid \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n)\mathbf{0} \equiv \mathbf{0}$	(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 .)

Example

$$\overbrace{A[msg[\langle M\rangle \mid out\ A.\ in\ B]]}^{Principal\ A} \mid \overbrace{B[open\ msg.\ (x).\ P]}^{Principal\ B}$$

$$A[\underbrace{msg[\langle M\rangle \mid out\ A.\ in\ B]}_{send\ M{:}A{=}{>}B}] \mid B[\underbrace{open\ msg.\ (x).\ P}_{receive\ x;\ P}]$$

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

Noticeable Inequivalences

- Replication creates new names:

$$!(\nu n)P \quad \not\equiv \quad (\nu n)!P$$

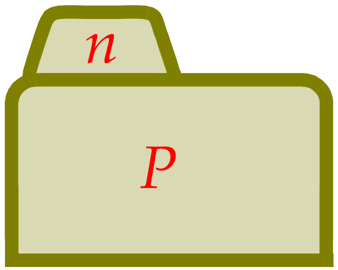
- Multiple n ambients have separate identity:

$$n[P] \,|\, n[Q] \quad \not\equiv \quad n[P \,|\, Q]$$

The Folder Calculus

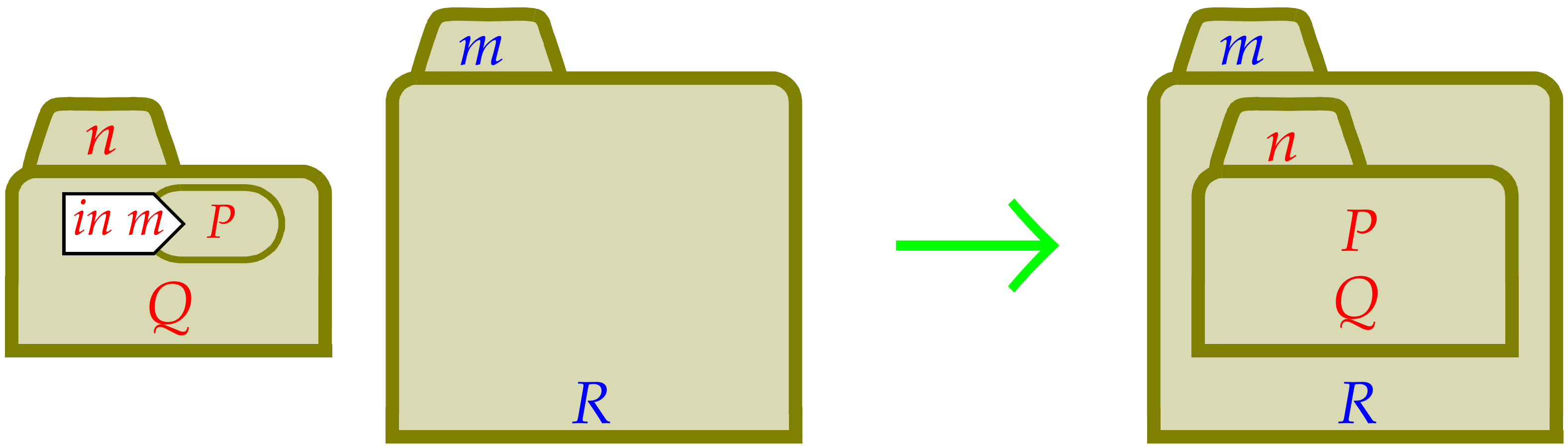
- Originally, a metaphor to explain the ambient calculus.
- Acquiring a life of its own:
  - ~ a “computationally complete multithreaded graphical office metaphor”.
  - ~ (optionally) typed.
  - ~ isomorphic to the ambient calculus.
- Disclaimer: not yet in the form of a useful graphical scripting language. But all the “useful” primitives are in principle expressible.

Folders

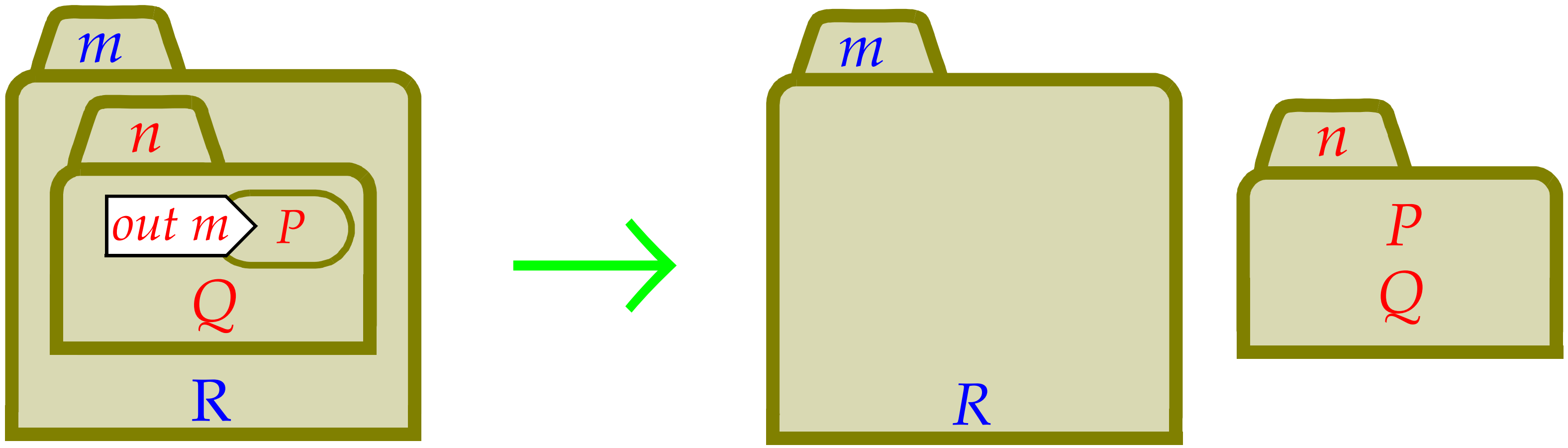


- A folder name n.
- Active contents P:
  - ~ hierarchical data and "gremlins".
  - ~ computational primitives for mobility and communication.

Enter Reduction

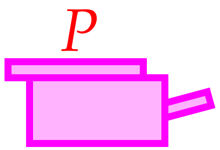


Exit Reduction

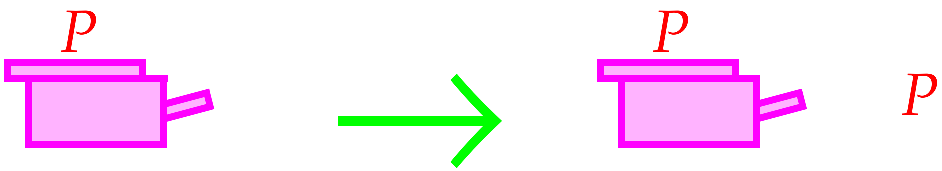


Open Reduction

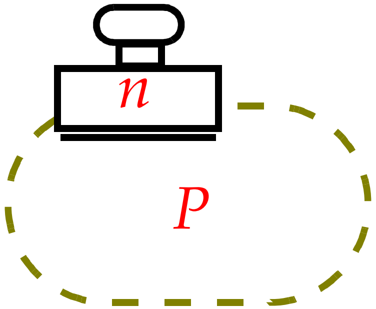
Copiers



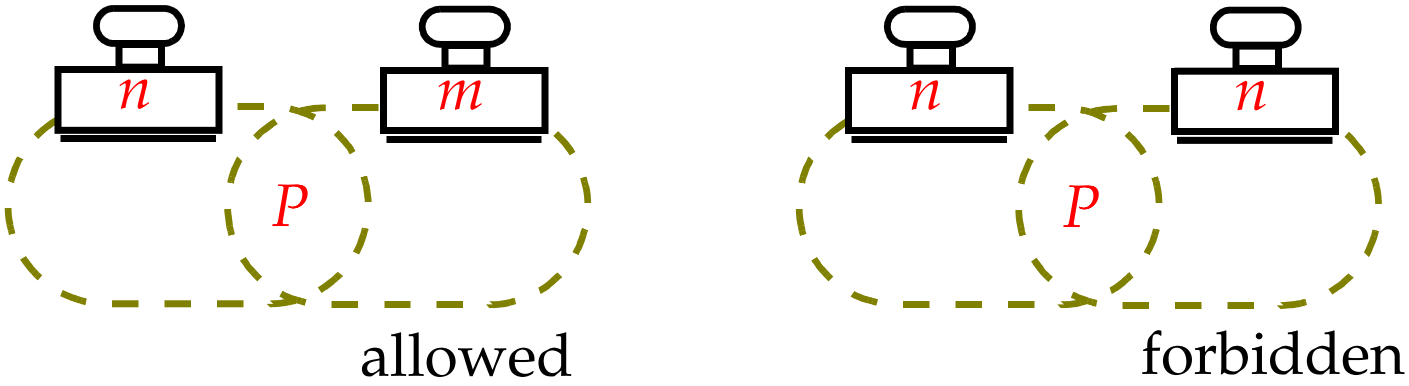
Copy Reduction



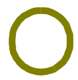
Rubber Stamps



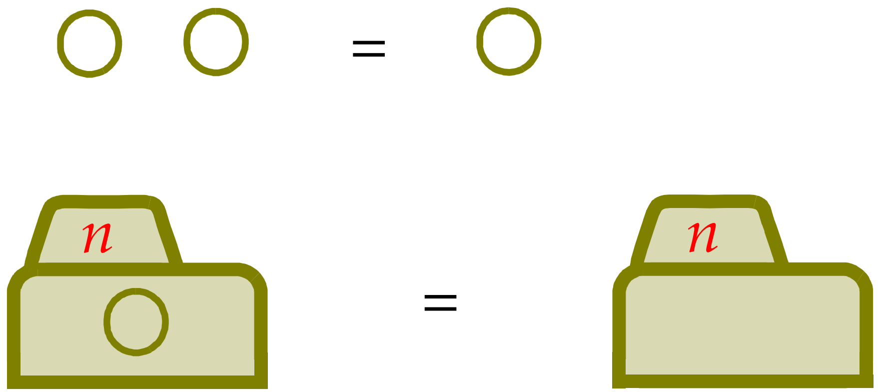
- Give authenticity to folders.
- Copiers are unable to accurately duplicate rubber stamps.



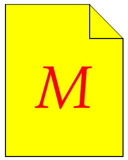
Leaves of the Syntax



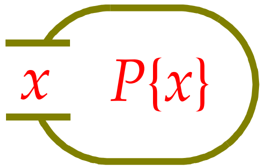
Inactive gremlin



Output and Input



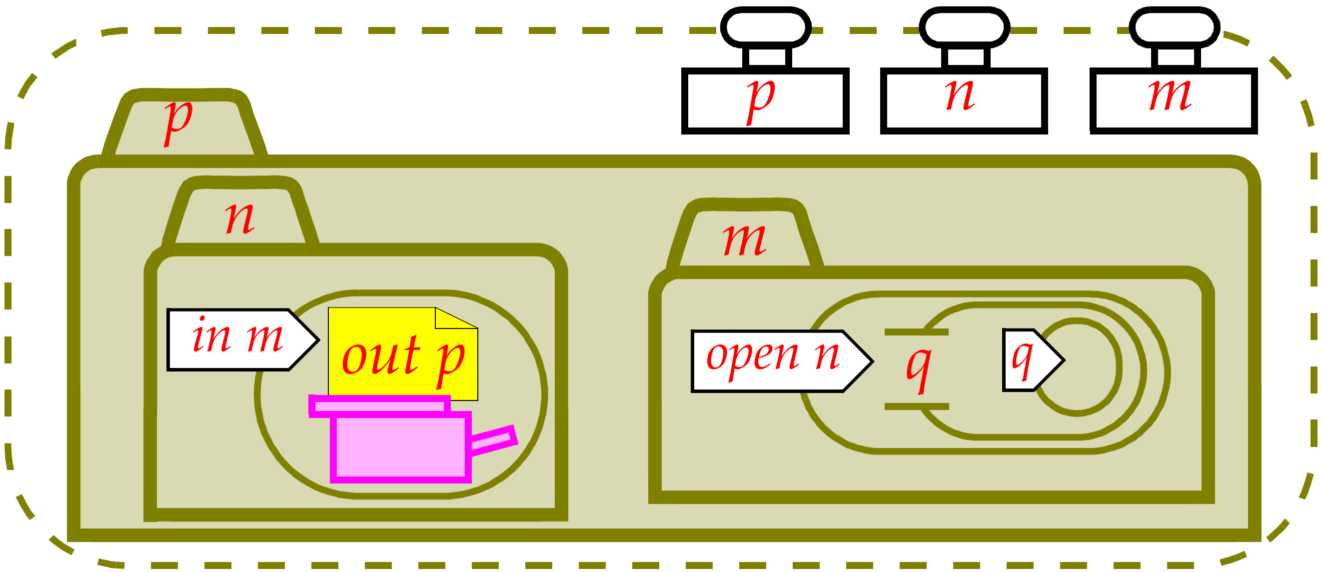
- A nameless file. (Originally: an asynchronous message.)



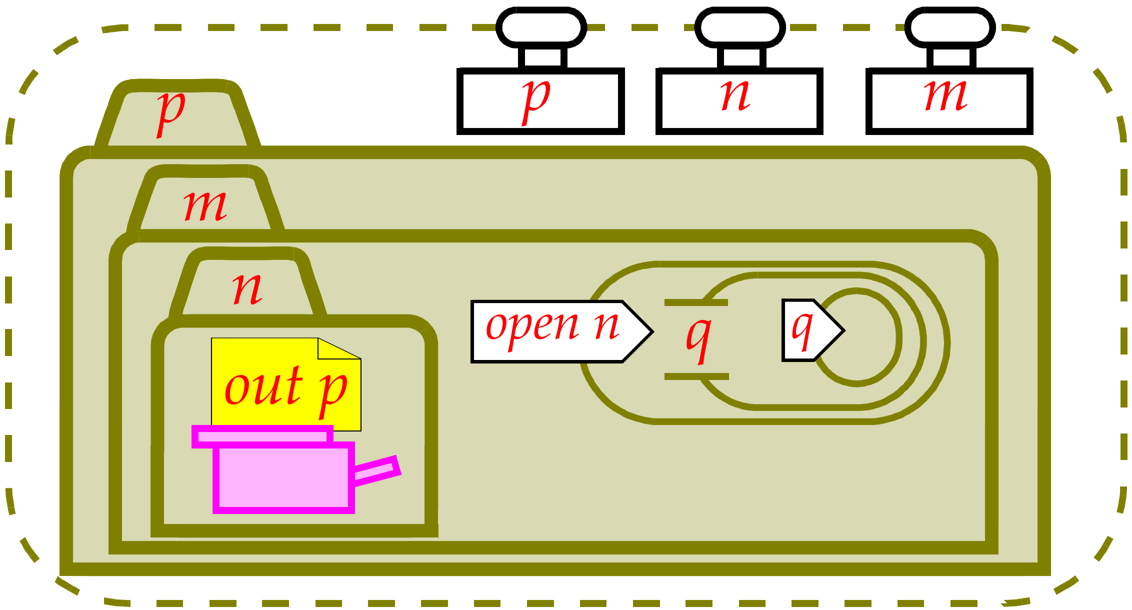
- A gremlin grabbing (reading and removing) a file.

Read Reduction

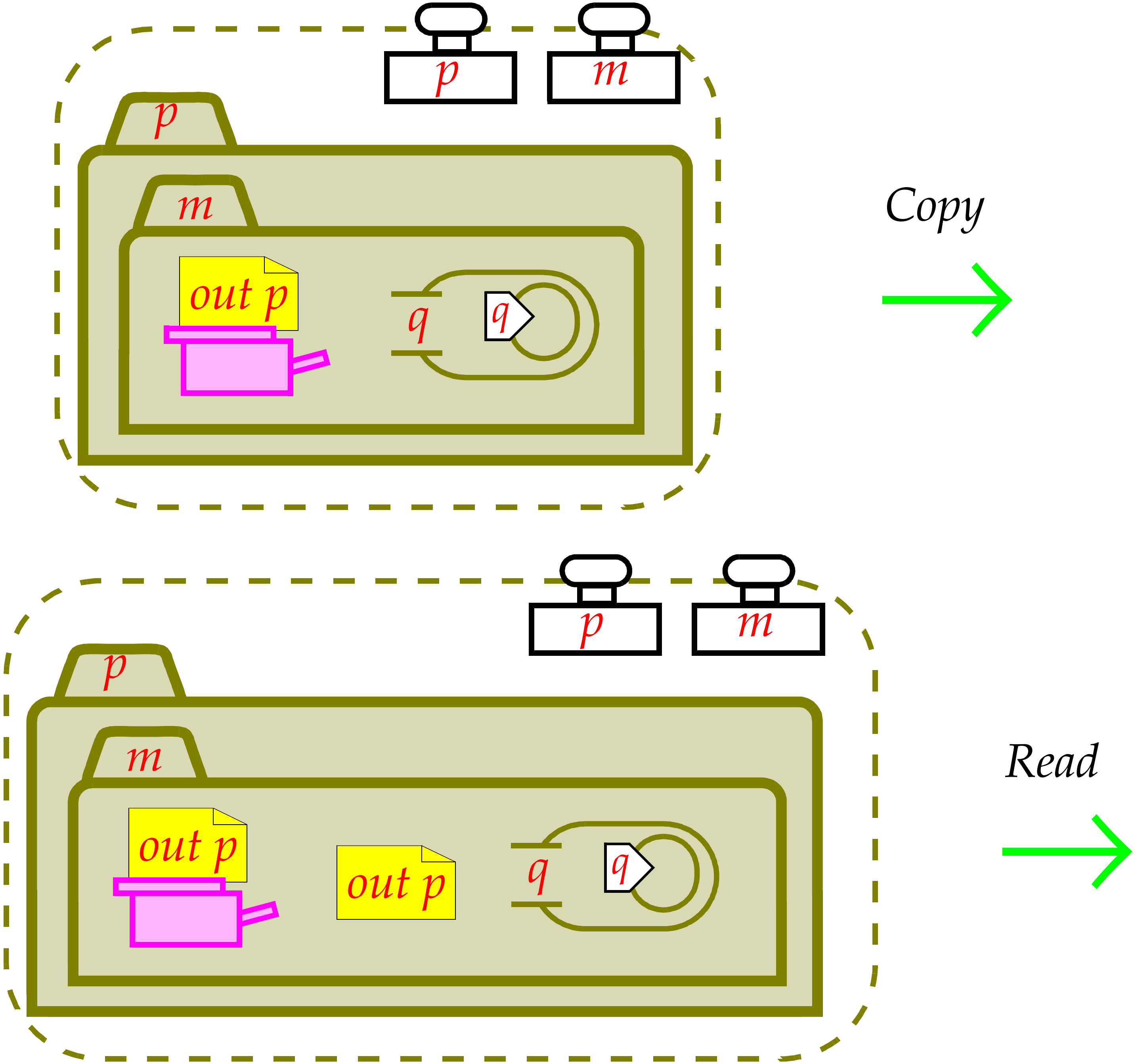
Example

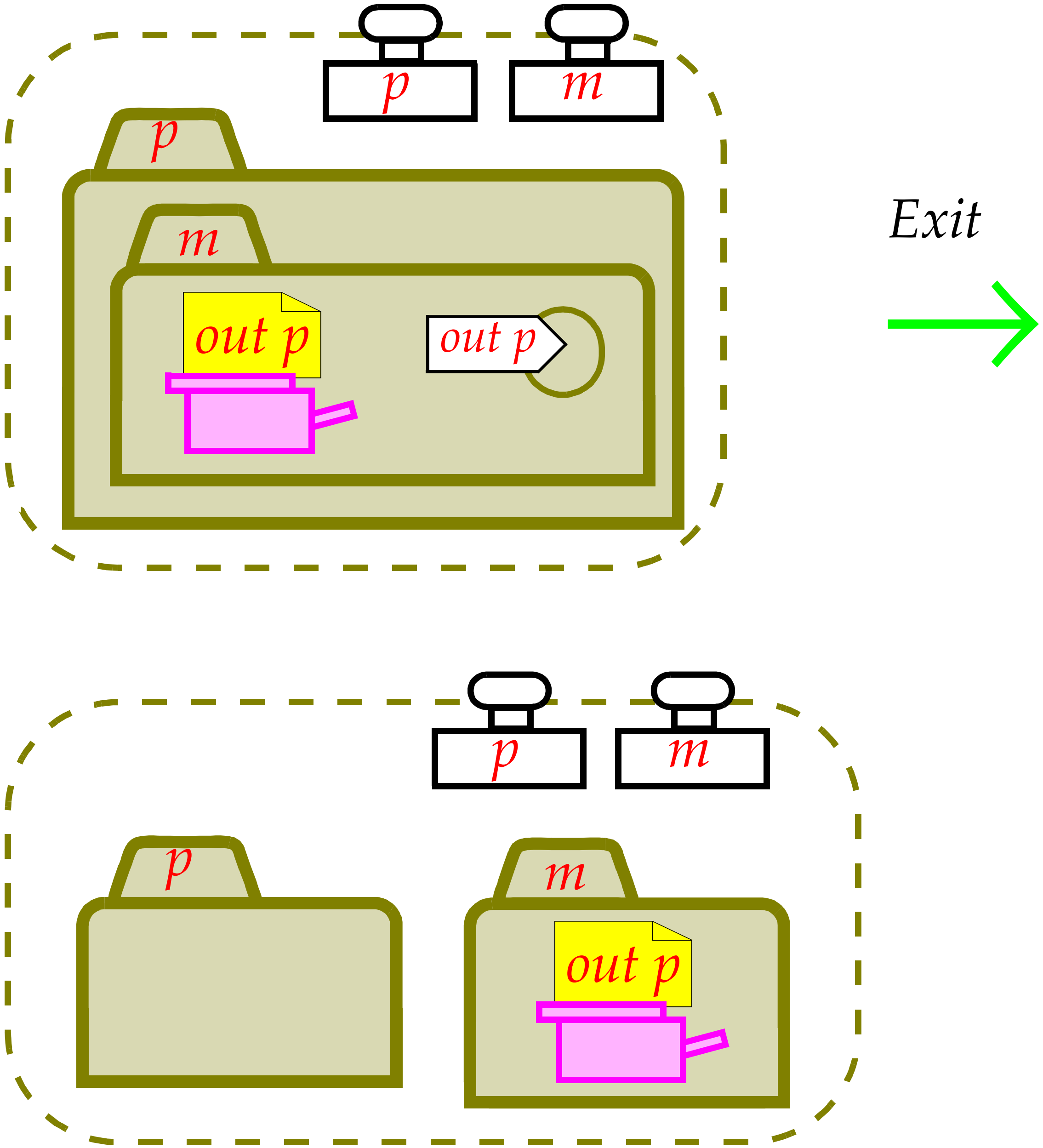


Enter

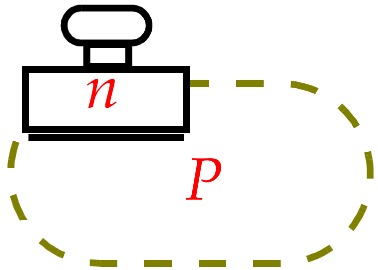
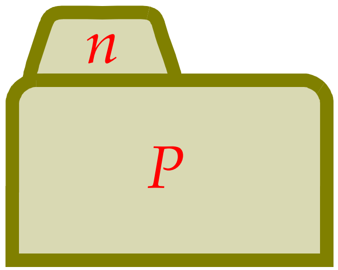
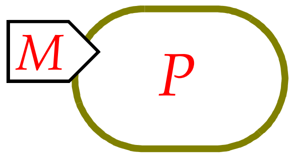


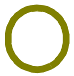
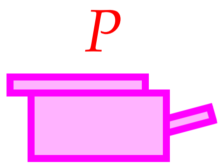
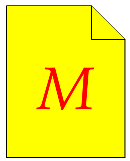
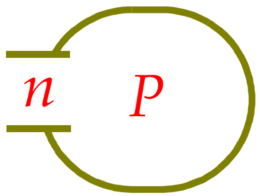
Open

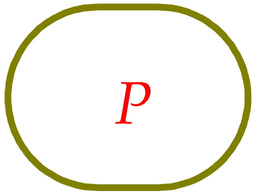

p
m
p
m
out p
q
q
Copy
p
m
p
m
out p
out p
q
q
Read


p
m
p
m
out p
out p
Exit
p
m
p
m
out p

Textual Syntax

Textual Syntax	*Visual Syntax*	*Comments*
$(\nu n)P$		New name n in scope P.
$n[P]$		Folder (ambient) of name n and contents P.
$M.P$		Action M followed by P.

$P \mid Q$	$P \quad Q$	Two processes in parallel. (Visually: contiguously placed in 2D.)
0		Inactive process (often omitted).
$!P$		Replication of P.
$\langle M \rangle$		Output M.
$(n).P$		Input n followed by P.

(P)		Grouping.

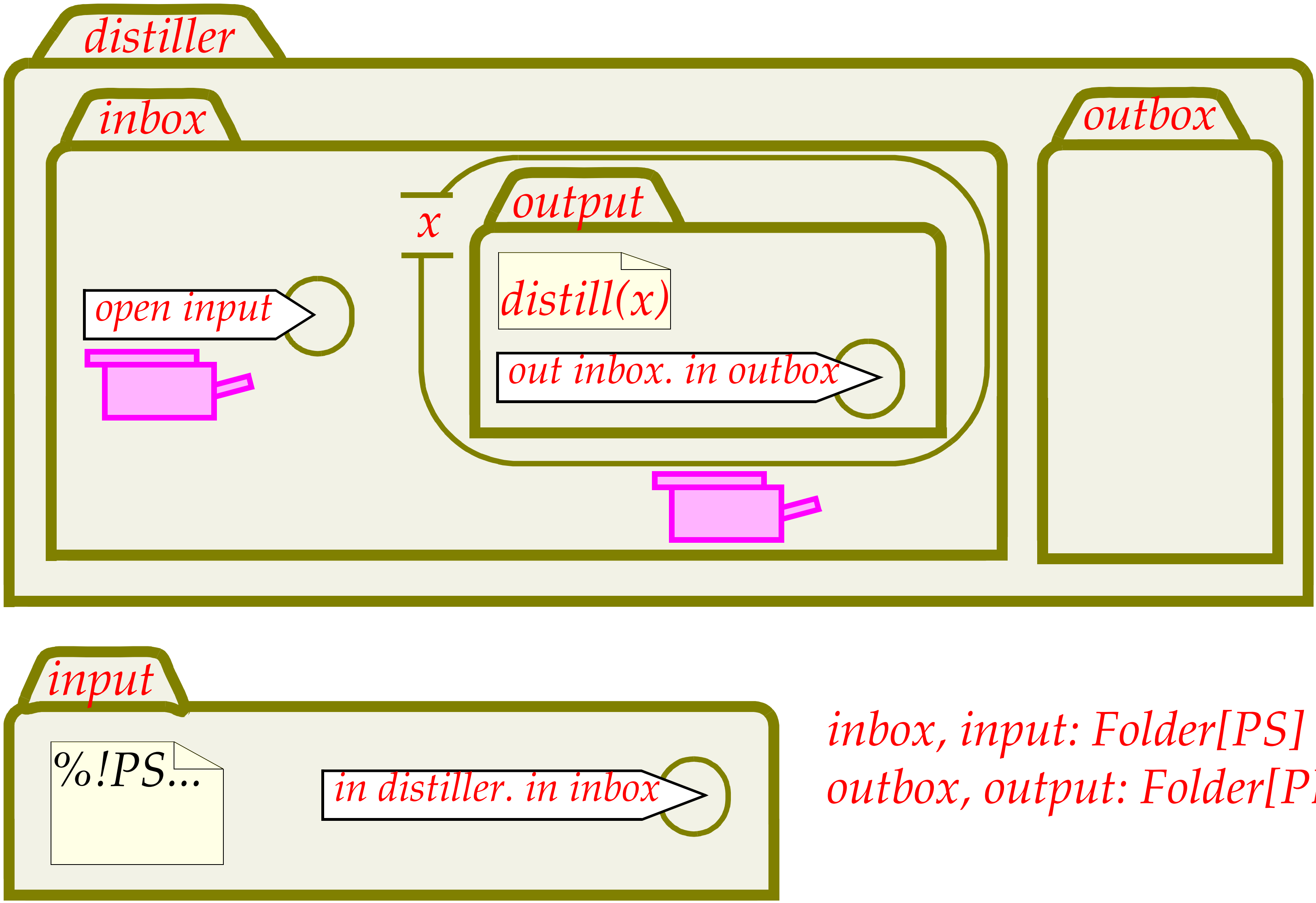
Data M,N

n	A name
in n	An entry capability
out n	An exit capability
open n	An open capability
here	The empty path of capabilities
$M.N$	The concatenation of two paths
" ... "	A string (file)
	etc.

Remarks

- The folder calculus is Turing-complete (even without the I/O operations), concurrent, with synchronization primitives.
- A type system can be used to make sure that each gremlin reads only messages of the appropriate type.
  - ~ The type of a file is associated with the name of the folder that contains it. All the files in a folder must have the same type.
  - ~ Subfolders of a given folder may contain files of different types.
  - ~ So we have a heterogeneous data hierarchy, but with well-typed I/O.

A Distiller Server

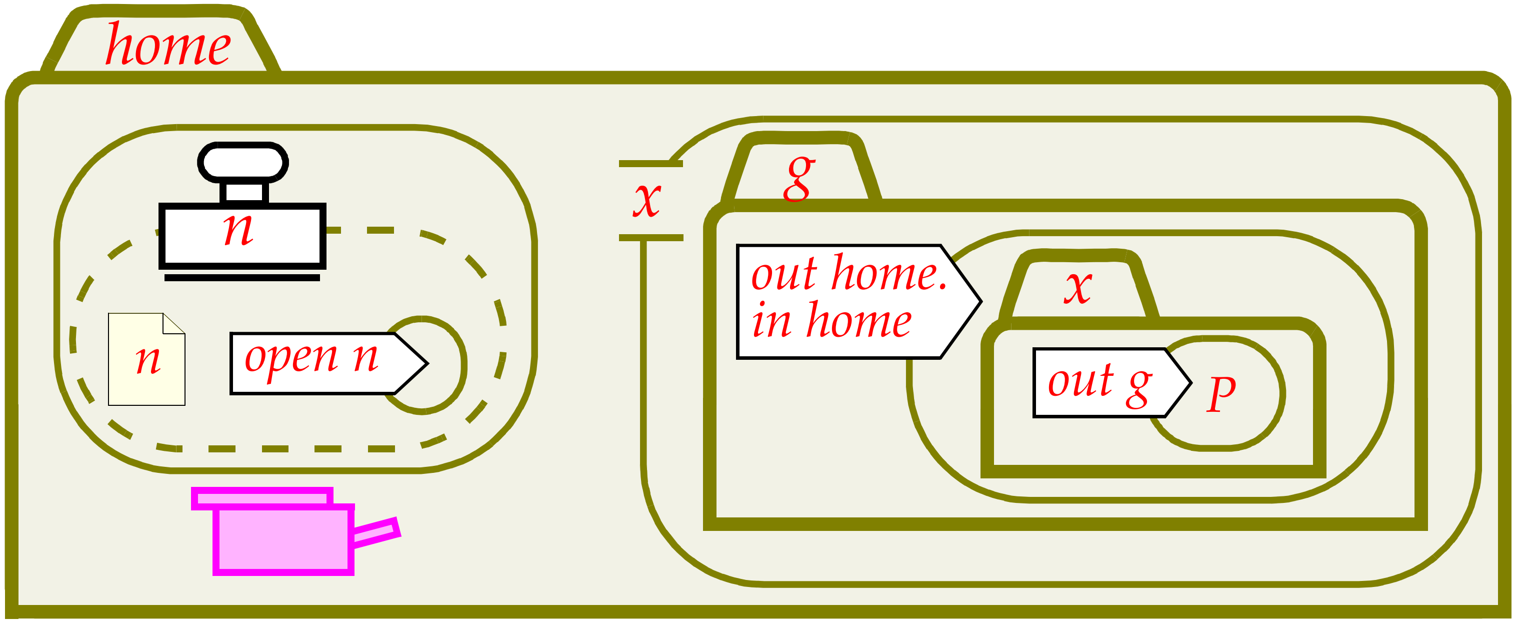


inbox, input: Folder[PS]
outbox, output: Folder[PDF]

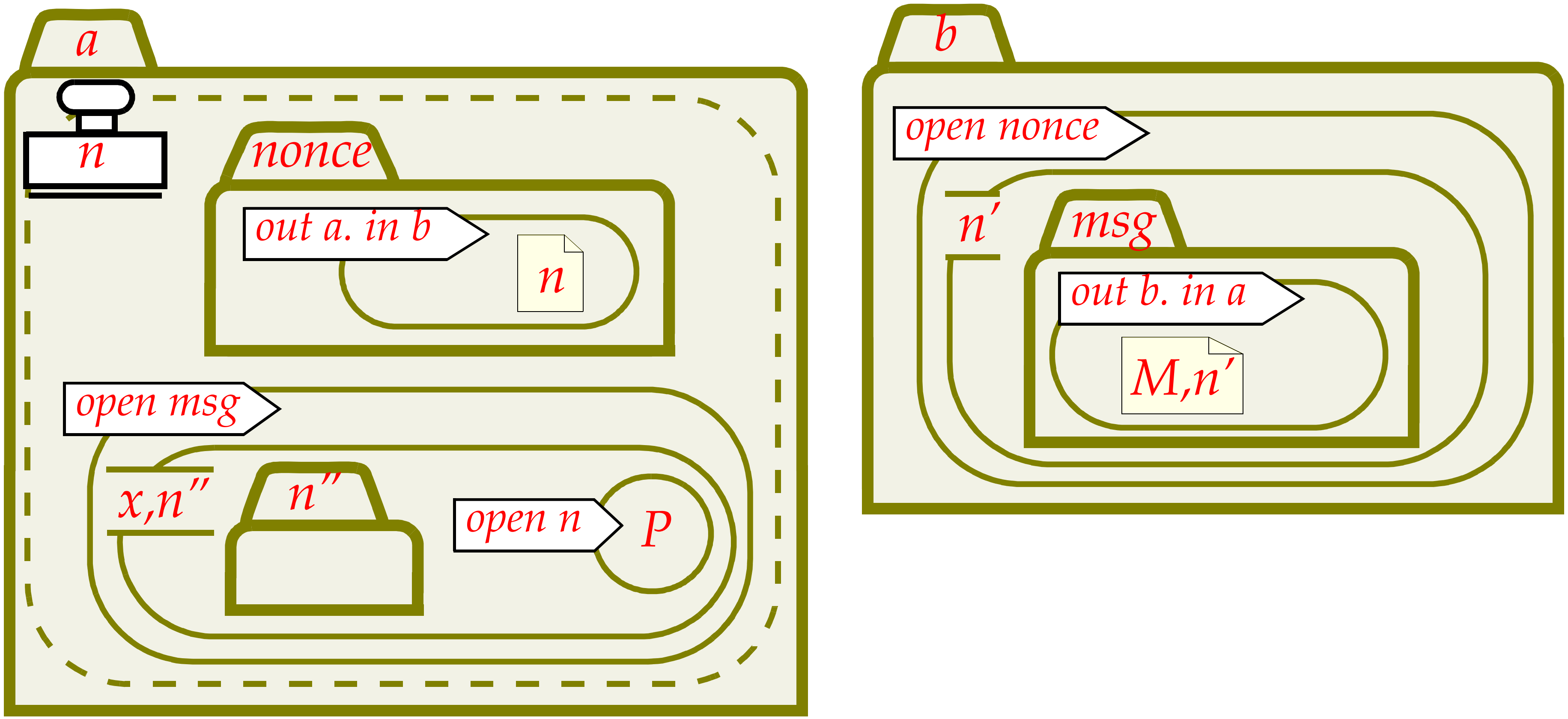
In textual form:

```
distiller[
  inbox[
    !open input |
    !(x) output[⟨distill(x)⟩ | out inbox. in outbox]] |
  outbox[]]
|
input[⟨"%!PS..."⟩ | in distiller. in inbox]
```

Authentication

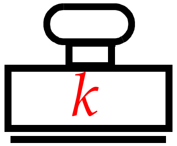


Nonces

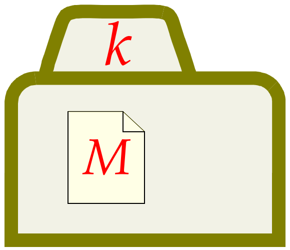


a
n
nonce
out a. in b
n
open msg
x,n″
n″
open n
P
b
open nonce
n′
msg
out b. in a
M,n′

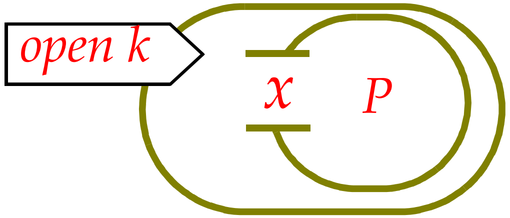
Shared Keys



generation of a fresh shared key k



encryption:
plaintext M inside a k-envelope



decryption:
opening a k-envelope and reading the contents

Expressiveness

- Some new features
  - ~ The primitives invented exclusively for process mobility end up being meaningful for security. (Various caveats apply.)
  - ~ The combination of mobility and cryptography in the same formal framework seems novel and intriguing.
  - ~ E.g., we can represent both mobility and (some) security aspects of “crossing a firewall”.

Expressiveness

- Old concepts that can be represented:
  - ~ Synchronization and communication mechanisms.
  - ~ Turing machines. (Natural encoding, no I/O required.)
  - ~ Arithmetic. (Tricky, no I/O required.)
  - ~ Data structures.
  - ~ π-calculus. (Easy, channels are ambients.)
  - ~ λ-calculus. (Hard, different than encoding λ in π.)
  - ~ Spi-calculus concepts. (Being debated.)

Expressiveness

- Net-centric concepts that can be represented:
  - ~ Named machines and services on complex networks.
  - ~ Agents, applets, RPC.
  - ~ Encrypted data and firewalls.
  - ~ Data packets, routing, active networks.
  - ~ Dynamically linked libraries, plug-ins.
  - ~ Mobile devices.
  - ~ Public transportation.

Ambients as Locks

- We can use *open* to encode locks:

$$\mathit{release}\ n.\ P \triangleq n[\,] \mid P$$
$$\mathit{acquire}\ n.\ P \triangleq \mathit{open}\ n.\ P$$

- This way, two processes can "shake hands" before proceeding with their execution:

$$\mathit{acquire}\ n.\ \mathit{release}\ m.\ P \mid \mathit{release}\ n.\ \mathit{acquire}\ m.\ Q$$

Turing Machines

$$
\begin{array}{l}
\mathit{end}[\mathit{extendLft} \mid S_0 \mid \\
\quad \mathit{square}[S_1 \mid \\
\qquad \mathit{square}[S_2 \mid \\
\qquad\quad \ldots \\
\qquad\qquad \mathit{square}[S_i \mid \mathit{head} \mid \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad \mathit{square}[S_{n\text{-}1} \mid \\
\qquad\qquad\qquad\quad \mathit{square}[S_n \mid \mathit{extendRht}]]\ ..\ ]\ ..\ ]]]
\end{array}
$$

Ambients as Mobile Processes

$$tourist \triangleq (x).\ joe[x.\ enjoy]$$
$$ticket\text{-}desk \triangleq\ !\langle in\ AF81SFO.\ out\ AF81CDG\rangle$$

$$SFO[ticket\text{-}desk \mid tourist \mid AF81SFO[route]]$$

$$\longrightarrow^* SFO[ticket\text{-}desk \mid$$
$$joe[in\ AF81SFO.\ out\ AF81CDG.\ enjoy] \mid$$
$$AF81SFO[route]]$$

$$\longrightarrow^* SFO[ticket\text{-}desk \mid$$
$$AF81SFO[route \mid joe[out\ AF81CDG.\ enjoy]]]$$

Ambients as Firewalls (buggy)

- Assume that the shared key k is already known to the firewall and the client.

$$Wally \triangleq (\nu\, w\, r)\, (\langle in\ r\rangle \mid r[open\ k.\ in\ w] \mid w[open\ r.\ P])$$
$$Cleo \triangleq (x).\ k[x.\ C]$$

$$Cleo \mid Wally$$
$$\longrightarrow^* (\nu\, w\, r)\, (\, (x).\ k[x.\ C] \mid \langle in\ r\rangle \mid r[open\ k.\ in\ w] \mid w[open\ r.\ P]\,)$$
$$\longrightarrow^* (\nu\, w\, r)\, (\, k[in\ r.\ C] \mid r[open\ k.\ in\ w] \mid w[open\ r.\ P]\,)$$
$$\longrightarrow^* (\nu\, w\, r)\, (\, r[k[C] \mid open\ k.\ in\ w] \mid w[open\ r.\ P]\,)$$
$$\longrightarrow^* (\nu\, w\, r)\, (\, r[C \mid in\ w] \mid w[open\ r.\ P]\,)$$
$$\longrightarrow^* (\nu\, w\, r)\, (\, w[r[C] \mid open\ r.\ P]\,)$$
$$\longrightarrow^* (\nu\, w)\ \ (\, w[C \mid P]\,)$$

Ambients as Firewalls

- Assume that the shared key k is already known to the firewall and the client.

$$
\begin{aligned}
\mathit{Wally} &\triangleq (\nu w)\,(k[\mathit{in}\ k.\ \mathit{in}\ w] \mid w[\mathit{open}\ k.\ P]) \\
\mathit{Cleo} &\triangleq k[\mathit{open}\ k.\ C]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Cleo} \mid \mathit{Wally} \\
&\quad \longrightarrow^* (\nu w)\,(\ k[\mathit{open}\ k.\ C] \mid k[\mathit{in}\ k.\ \mathit{in}\ w] \mid w[\mathit{open}\ k.\ P]\ ) \\
&\quad \longrightarrow^* (\nu w)\,(\ k[k[\mathit{in}\ w] \mid \mathit{open}\ k.\ C] \mid w[\mathit{open}\ k.\ P]\ ) \\
&\quad \longrightarrow^* (\nu w)\,(\ k[\mathit{in}\ w \mid C] \mid w[\mathit{open}\ k.\ P]\ ) \\
&\quad \longrightarrow^* (\nu w)\quad w[k[C] \mid \mathit{open}\ k.\ P] \\
&\quad \longrightarrow^* (\nu w)\quad w[C \mid P]
\end{aligned}
$$

Comments

- One secret names is introduced: w is the secret name of the firewall.
- We want to verify that Cleo knows the key k: this is done by *in* k. After that, Cleo gives control to *in* w to enter the firewall.

The Asynchronous π-calculus

- A named channel is represented by an ambient.
  - The name of the channel is the name of the ambient.
  - Communication on a channel is becomes local I/O inside a channel-ambient.
  - A conventional name, *io*, is used to transport I/O requests into the channel.

$$\begin{array}{lcl} (ch\ n)P & \triangleq & (\nu n)\ (n[!open\ io] \mid P) \\ n(x).P & \triangleq & (\nu p)\ (io[in\ n.\ (x).\ p[out\ n.\ P]] \mid open\ p) \\ n\langle M\rangle & \triangleq & io[in\ n.\ \langle M\rangle] \end{array}$$

- These definitions satisfy the expected reduction:

$$n(x).P \mid n\langle M\rangle \quad \longrightarrow^* \quad P\{x{\leftarrow}M\}$$

in presence of a channel for n.

- Therefore:

$$
\begin{array}{lcl}
\langle\!\langle(\nu n)P\rangle\!\rangle & \triangleq & (\nu n)\;(n[!open\;io] \mid \langle\!\langle P\rangle\!\rangle) \\
\langle\!\langle n(x).P\rangle\!\rangle & \triangleq & (\nu p)\;(io[in\;n.\;(x).\;p[out\;n.\;\langle\!\langle P\rangle\!\rangle]] \mid open\;p) \\
\langle\!\langle n\langle m\rangle\rangle\!\rangle & \triangleq & io[in\;n.\;\langle m\rangle] \\
\langle\!\langle P \mid Q\rangle\!\rangle & \triangleq & \langle\!\langle P\rangle\!\rangle \mid \langle\!\langle Q\rangle\!\rangle \\
\langle\!\langle !\mathrm{P}\rangle\!\rangle & \triangleq & !\langle\!\langle \mathrm{P}\rangle\!\rangle
\end{array}
$$

- ~ The choice-free synchronous π-calculus, can be encoded within the asynchronous π-calculus.
- ~ The λ-calculus can be encoded within the asynchronous π-calculus.

Contextual Equivalence

- Exhibition

$$P{\downarrow}n \quad \Leftrightarrow \quad P \equiv (\nu n_1 \dots n_p)(n[Q] \mid R) \;\wedge\; n \notin \{n_1 \dots n_p\}$$

- Convergence

$$P{\Downarrow} \quad \Leftrightarrow \quad \exists n.\; P \longrightarrow^* Q \;\wedge\; Q{\downarrow}n$$

- Contextual Equivalence

$$P \approx Q \quad \Leftrightarrow \quad \forall \boldsymbol{C}\{\!\!\{\bullet\}\!\!\}.\; \boldsymbol{C}\{\!\!\{P\}\!\!\}{\Downarrow} \Leftrightarrow \boldsymbol{C}\{\!\!\{Q\}\!\!\}{\Downarrow}$$

Firewalls

- $n[P]$ is a firewall named n protecting P.
- $in\ n$ is the capability needed to enter the firewall.
- $out\ n$ is the capability needed to exit the firewall.
- The *context* is the Internet.

- The Perfect-Firewall Equation:

$$(\nu n)\, n[P] \approx \mathbf{0} \qquad (\text{if } n \text{ not in } P)$$

Cryptography

- The ambient calculus can, without special extensions, model certain cryptographic procedures.
  - ~ In particular, it can model the most basic subset of the spi-calculus:

{M}N	shared-key encryption of M by N
decrypt M with N	shared-key decryption

- It does not embrace a particular implementation:
  - ~ It does not model the ability an attacker may have to compare bit patterns.
  - ~ It does not model the ability an attacker may have to exploit properties of a specific underlying crypto.

Nonces

- A nonce is simply a fresh name that can, for example, be communicated by an output action.

$$Q \mid (\nu n)\,(\langle n \rangle \mid P) \qquad \text{output a nonce } n \text{ for } Q$$

When the nonce comes back to P, it can be verified by *open* n.

Shared Keys

- A name can be used as a shared key, as long as it is kept secret and shared only by certain parties.

$k[\langle txt \rangle]$	encrypt *txt* with the shared key *k*
open $k.\ (x).\ P$	decrypt with the shared key *k* and read the message

- Anybody who knows *k* can decrypt a message $k[\langle txt \rangle]$:
  - ~ Either by *open k* (destructively).
  - ~ Or by *in k* followed by *out k* (non-destructively).

Public Keys: Signed Messages

- If $k[\langle txt \rangle]$ is the plaintext *txt* encrypted by k, then *open* k represents the (public) ability to open a k-envelope, without knowing k.

Principal A	
(νk)	create a new signature key
$!\langle open\ k \rangle$	publish the signature verifier
$\mid k[\langle txt \rangle]$	sign a message
Principal B	
$(open\text{-}cap).$	acquire the signature verifier
$open\text{-}cap.$	verify an available message
$(msg).\ P$	read the message and proceed

Public Keys: Coded Message

- If $k[\langle txt \rangle]$ is the plaintext *txt* encrypted by k, then $(x).\ k[\langle x \rangle]$ represents the (public) ability to insert a plaintext in a k-envelope, without knowing k.

Principal A

(νk)	create a new encryption key
$!(x).\ k[\langle x \rangle]$	publish message encryptors
$\mid\ !open\ k.\ (x).\ P$	decrypt incoming messages and proceed

Principal B

$\langle txt \rangle$	encrypt a message for A
	(assuming an encryptor for A is available here)
	(possibly route it back to A)

Ciphers

- $k[\langle txt \rangle]$ is the plaintext txt encrypted with key k.
- $P \approx Q$ means "no attacker can tell P from Q".
- The Perfect-Cipher Equation:

$$(\nu k_1)\, k_1[\langle txt_1 \rangle] \;\approx\; (\nu k_2)\, k_2[\langle txt_2 \rangle]$$

  ~ Simply because $(\nu k_1)\, k_1[\langle txt_1 \rangle] \approx \mathbf{0} \approx (\nu k_2)\, k_2[\langle txt_2 \rangle]$.

  ~ This is a consequence of (a) the reductions allowed in the calculus, (b) the absence of other reductions that might make distinctions, (c) the (debatable) interpretation of ambient operations as crypto operations.

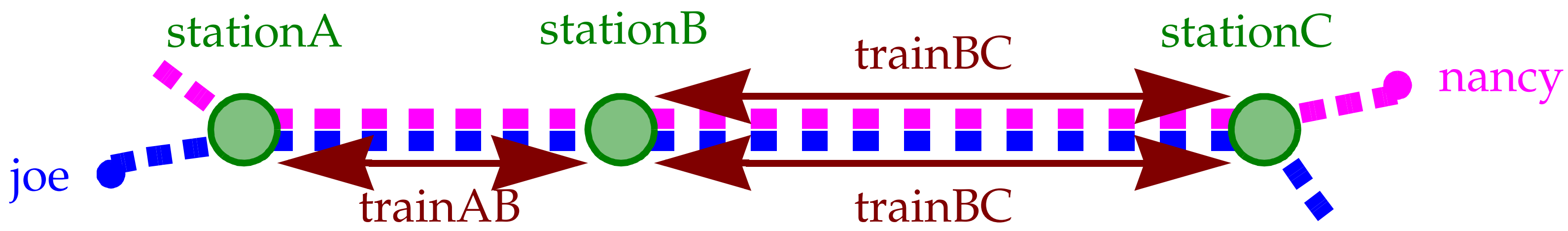
Calculi vs. Reality

- Calculi make “implicit security assumptions”.
  - *Nominal calculi*, like π, spi, assume that <u>nobody</u> can guess the name of a private channel.
  - The ambient calculus assumes that <u>nobody</u> can extract a name from a capability.
  - Consequences include the perfect-cipher equation.
- A) This is good.
  - These assumptions are “security abstraction” that enable high-level reasoning (via $\approx$).
  - These assumptions can be realized by different implementation (crypto) techniques.
  - They may increase practical security by providing a programming model that is more transparent.

- B) This is bad.
  - Such assumption are dangerous since they are not obviously "realistic". How do they map to algebraic properties of the underlying crypto primitives?
  - They may hold within the calculus, but do they keep holding under low-level attacks (if somebody can dissect an agent)?
- (Speculation.) Implicit security assumptions must be made explicit and must be "securely implemented".
  - One must describe an implementation of the calculus in terms of realistic cryptographic primitives.
  - One must prove that the implementation is (1) correct and (2) prevents certain low-level attacks. [Abadi, Gonthier, Fournet]

Language Applications

Transportation




```
let train(stationX stationY XYatX XYatY tripTime) =
  new moving.        // assumes the train originates inside stationX
    moving[rec T.
      be XYatX. wait 2.0.
      be moving. go out stationX. wait tripTime. go in stationY.
      be XYatY. wait 2.0.
      be moving. go out stationY. wait tripTime. go in stationX.
      T];

new stationA stationB stationC ABatA ABatB BCatB BCatC.
  stationA[ train(stationA stationB ABatA ABatB 10.0) ] |
  stationB[ train(stationB stationC BCatB BCatC 20.0) ] |
  stationC[ train(stationC stationB BCatC BCatB 30.0) ] |
```

```
new joe.
  joe[
    go in stationA.
    go in ABatA. go out ABatB.
    go in BCatB. go out BCatC.
    go out stationC] |

new nancy.
  nancy[
    go in stationC.
    go in BCatC. go out BCatB.
    go in ABatB. go out ABatA.
    go out stationA]
```

Execution trace

```
moving: Be ABatA
moving: Be BCatC
moving: Be BCatB
nancy: Moved in stationC
nancy: Moved in BCatC
joe: Moved in stationA
joe: Moved in ABatA
ABatA: Be moving
BCatC: Be moving
moving: Moved out stationC
BCatB: Be moving
moving: Moved out stationB
moving: Moved out stationA
moving: Moved in stationB
moving: Be ABatB
joe: Moved out ABatB
ABatB: Be moving
moving: Moved out stationB
moving: Moved in stationC
moving: Be BCatC
BCatC: Be moving
moving: Moved out stationC
moving: Moved in stationA
moving: Be ABatA
ABatA: Be moving
moving: Moved out stationA
moving: Moved in stationB
```

```
moving: Be BCatB
nancy: Moved out BCatB
joe: Moved in BCatB
BCatB: Be moving
moving: Moved out stationB
moving: Moved in stationB
moving: Be ABatB
nancy: Moved in ABatB
ABatB: Be moving
moving: Moved out stationB
moving: Moved in stationB
moving: Be BCatB
BCatB: Be moving
moving: Moved out stationB
moving: Moved in stationA
moving: Be ABatA
nancy: Moved out ABatA
nancy: Moved out stationA
ABatA: Be moving
moving: Moved out stationA
moving: Moved in stationB
moving: Be ABatB
moving: Moved in stationC
moving: Be BCatC
joe: Moved out BCatC
joe: Moved out stationC
moving: Moved in stationC
...
```

Wide Area Languages

- The ambient/folder calculus is a minimal formalism designed for theoretical study. As such, it is not a "programming language".
- Still, the ambient calculus is designed to match fundamental WAN characteristics.
- We now discuss how ambient characteristics might look like when extrapolated to a programming language.

Barriers

- Mobility is all about barriers:
  - ~ Locality = **barrier topology**.
  - ~ Process mobility = **barrier crossing**.
  - ~ Security = **(in)ability to cross barriers**.
  - ~ Communication = **interaction within a barrier.**
  - ~ No immediate (un-mediated) action at a distance (across barriers).
- Ambients embed this barrier-based view of mobility (extrapolated from Telescript), which is grounded on WAN observables.
- A "wide-area language" is one that does not contain features violating this view of computation.

Ambients as a Programming Abstraction

- Our basic abstraction is that of mobile computational ambients.
- The ambient calculus brings this abstraction to an extreme, by representing *everything* in terms of ambients at a very fine grain.
- In practice, ambients would have to be medium or large-grained entities. Ambient contents should include standard programming subsystems such as modules, classes, objects, and threads.
- But: the ability to smoothly move a collection of running threads is almost unheard of in current software infrastructures. Ambients would be a novel and non-trivial addition to our collection of programming abstractions.

Names vs. Pointers

- The only way to denote an ambient is by its name.
  - ~ One may possess a name without having immediate access to any ambient of that name (unlike pointers).
  - ~ Name references are never “broken” but may be “blocked” until a suitable ambient becomes available.
- Uniformly replace pointers (to data structures etc.) by names.
  - ~ At least across ambient boundaries.
  - ~ This is necessary to allow ambients to move around freely without being restrained by immobile ties.

Locations

- Ambients can be used to model both physical and virtual locations.
  - ~ Some physical locations are mobile (such as airplanes) while others are immobile (such as buildings).
  - ~ Similarly, some virtual locations are mobile (such as agents) while others are immobile (such as mainframe computers).
- Mobility distinctions are not part of the basic semantics of ambients.
  - ~ Can be added as a refinement of the basic model, or
  - ~ Can be embedded in type systems that restrict the mobility of certain ambients.

Migration and Transportation

- Ambients offer a good paradigm for application migration.
  - ~ If an ambient encloses a whole application, then the whole running application can be moved without need to restart it or reinitialize it.
  - ~ In practice, an application will have ties to the local window system, the local file system, etc. These ties, however, should only be via ambient names.
  - ~ After movement the application can smoothly move and reconnect its bindings to the new local environment. (Some care will still be needed to restart in a good state).

Communication

- The communication primitives of the ambient calculus (local to an ambient) do not support global consensus or failure detection.
- These properties should be preserved by any higher-level communication primitives that may be added to the basic model, so that the intended semantics of communication over a wide-area networks is preserved.
  - ~ RPC, interpreted as mobile packets that transport and deposit messages to remote locations.
  - ~ Parent-child communication
  - ~ Communication between siblings.

Synchronization

- The ambient calculus is highly concurrent.
  - It has high-level synchronization primitives that are natural and effective (as shown in the examples).
  - It is easy to represent basic synchronization constructs, such as mutexes:

$$\begin{array}{lcll} \mathit{release}\ n; P & \triangleq & n[\,] \mid P & \text{release a mutex called } n\text{, and} \\ \mathit{acquire}\ n; P & \triangleq & \mathit{open}\ n.\ P & \text{acquire a mutex called } n\text{, ther} \end{array}$$

- Still, additional synchronization primitives are desirable.
  - A useful technique is to synchronize on the change of name of an ambient:

$$n[\mathit{be}\ m.P \mid Q] \;\rightarrow\; m[P \mid Q]$$

  - (See also the Seal calculus by Castagna and Vitek.)

Static and Dynamic Binding

- The names of the ambient calculus represent an unusual combination of static and dynamic binding.
  - ~ The names obey the classical rules of static scoping, including consistent renaming, capture-avoidance, and block nesting.
  - ~ The navigation primitives behave by dynamically binding/ linking a name to any ambient that has the right name.
- Definitional facilities can similarly be derived in static or dynamic binding style. E.g.:
  - ~ Statically bound function definitions.
  - ~ Dynamically bound resource definitions.

Modules

- An ambient containing definitions is similar to a module/class.
  - ~ Remote invocation is like qualified module access.
  - ~ *open* is like inheritance.
  - ~ *copy* is like object generation from a prototype.
- Unusual "module" features:
  - ~ Ambients are *first class modules*: one can choose at run time which particular instance of a module to use.
  - ~ Ambients support *dynamic linking*: missing subsystems can be added to a running system by placing them in the right spot.
  - ~ Ambients support *dynamic reconfiguration*. The identity of individual modules is maintain at run time. The blocking semantics allows smooth suspension and reactivation. The hierarchical structure allows replacement of subsystems.

Security

- Ambient security is based on boundaries and capabilities, as opposed to a cryptography, or access-control.
- These three models are all interdefinable. In our case:
  - ~ Access control is obtained by using ambients to implement RPC-like invocations that have to cross boundaries and authenticate every time.
  - ~ Cryptography is obtained by interpreting ambient names (by assumption unforgeable) as encryption keys.
- The ambient security model is high level.
  - ~ It maps naturally to administrative domains and sandboxes.
  - ~ It allows the direct discussion of virus, trojan horses, infection of mobile agents, firewall crossing, etc.

Summary of WAL Features

- No “hard” pointers.

  Remote references are URLs, symbolic links, or such.

- Migration/Transportation

  Thread migration.

  Data migration.

  Whole-application migration.

- Dynamic linking.

  A missing library or plug-in may suddenly show up.

- Patient communication.

  Blocking/exactly-once semantics.

- Built-in security primitives.

Current Progress

- An informal paper describing wide-area computation, the Folder Calculus, and ideas for wide-area languages.
- A technical paper about the basic Ambient Calculus.
- A technical paper about techniques for proving equational properties of Ambients.
- A technical paper about a type systems for Ambients (“Exchange Types”) regulating communication.
- Work in progress with Giorgio Ghelli about type systems for regulating mobility.
- A Java applet implementation of the Ambient Calculus, and a tech report about its thread synchronization algorithm.

www.luca.demon.co.uk

Conclusions

- The notion of *named, active, hierarchical, mobile ambients* captures the structure and properties of wide-area networks and of mobile computing and computation.
- The ambient calculus formalizes ambient notions simply and powerfully.
  - ~ It is no more complex than common process calculi.
  - ~ It supports reasoning about mobility and security.
- It provides a basis for envisioning new programming methodologies, libraries, and languages for wide-area computation.