Part 3

Ambient Types

Luca Cardelli
Giorgio Ghelli
Andy Gordon

Types for Exchange Control

- Ambients exchange information by reading an writing to the “local ether”. In an untyped system, the ether can be full of garbage.
- How do we make sure that ether interactions are well typed? We need to track the exchanges of messages between processes.

Like Typing a File System

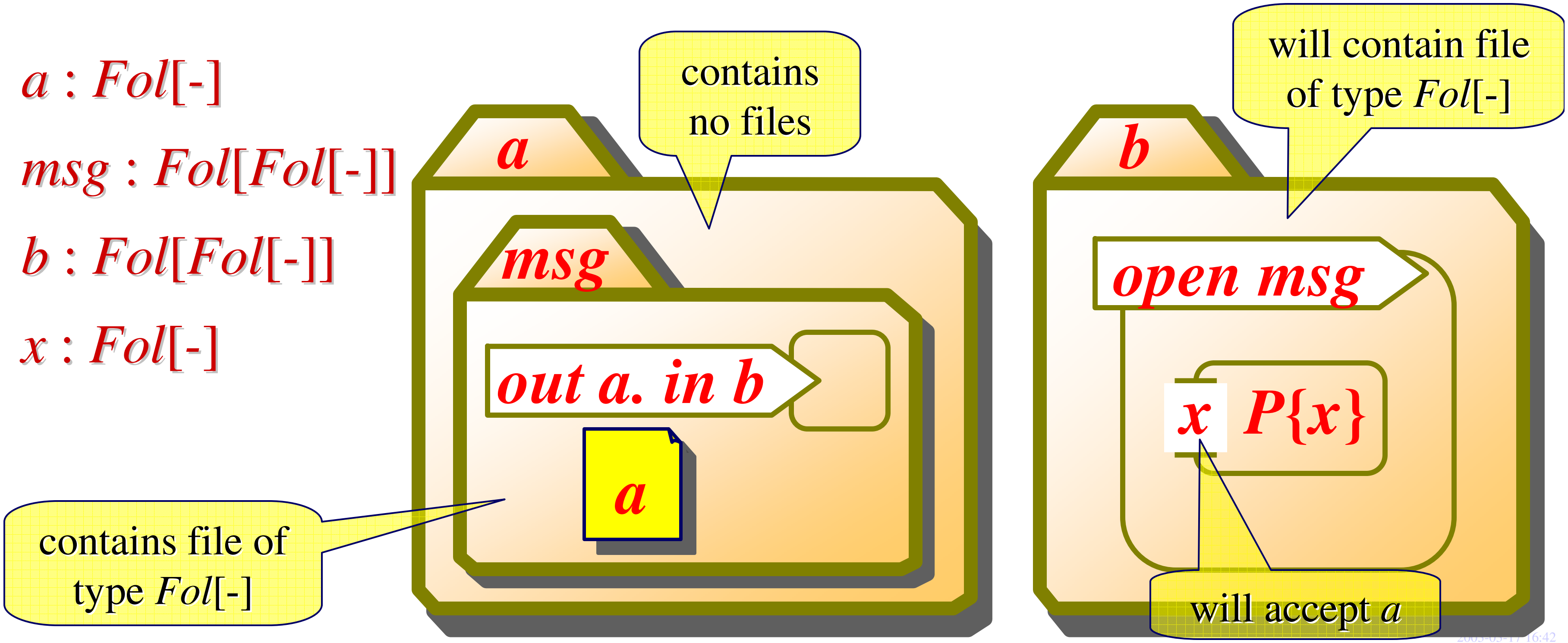
- $n : Fol[T]$ means that n is a name for folders that can contain only files of type T. E.g.: $ps : Fol[Postscript]$.
- Nothing is said about the subfolders of folders of name n: they can have any name and any type (and can come and go).
- Hierarchy rearrangements are totally unconstrained.

$a : Fol[\text{-}]$

$msg : Fol[Fol[\text{-}]]$

$b : Fol[Fol[\text{-}]]$

$x : Fol[\text{-}]$



Need for Distinctions

- The ambients syntax does not distinguish between names and capabilities, therefore it permits strange terms like:

  $in\ n[P]$ (stuck)

  $n.P$ (stuck)

- This cannot be avoided by a more precise syntax, because such terms may be generated by interactions:

  $\langle in\ n\rangle \mid (m).m[P] \longrightarrow in\ n[P]$

  $\langle n\rangle \mid (m).m.P \longrightarrow n.P$

  $(m).\ (m.P \mid m[Q])$ (tests whether m is a name or a capability!)

- We have two sorts of things (ambient names and capabilities) that we want to use consistently. A type system should do the job.

- Desired property: a well-typed program does not produce insane terms like $in\ n[P]$ and $n.P$.

Exchange Types

$W ::=$	message types
$Amb[T]$	ambient name allowing T exchanges
$Cap[T]$	capability unleashing T exchanges
$T ::=$	process types
Shh	no exchange
$W_1 \times \ldots \times W_k$	tuple exchange (**1** is the null product)

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

Polyadic Ambient Calculus

$P,Q ::=$

$(\nu n{:}W)P$

typed binder

0

$P \mid Q$

$!P$

$M[P]$

$M.P$

$(n_1{:}W_1, \ldots, n_k{:}W_k).P$

typed binders

polyadic input

$\langle M_1, \ldots, M_k \rangle$

polyadic output

$M,N ::=$

n

$in\ M$

$out\ M$

$open\ M$

ε

$M.N$

Reduction

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

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

$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$

$(n_1{:}W_1, \ldots, n_k{:}W_k).P \mid \langle M_1, \ldots, M_k \rangle \longrightarrow P\{n_1 \leftarrow M_1, \ldots, n_k \leftarrow M_k\}$

type oblivious

$P \longrightarrow Q \quad \Rightarrow \quad (\nu n{:}W)P \longrightarrow (\nu n{:}W)Q$

type oblivious

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

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

$P' \equiv P,\ P \longrightarrow Q,\ Q \equiv Q' \quad \Rightarrow \quad P' \longrightarrow Q'$

Structural Congruence

- As usual (polyadic).

Intuitions: Typing of Processes

- If M is a W, then $\langle M \rangle$ is a process that exchanges W.

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

- If P is a process that may exchange W, then $(n{:}W).P$ is too.

  $P : W \Rightarrow (n{:}W).P : W$

- If P and Q are processes that may exchange T, then $P \mid Q$ is too. (Similarly for $!P$.)

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

- Both $\mathbf{0}$ and $n[P]$ exchange nothing at the current level, so they can have any exchange type, and can be added in parallel freely.

- Therefore, W-inputs and W-outputs are tracked so that they match correctly when placed in parallel.

Intuitions: Typing of Open

- We have to worry about *open*, which might open-up a *T*-ambient and unleash *T*-exchanges inside an *S*-ambient.
- We decorate each ambient name with the *T* that can be exchanged in ambients of that name. Different ambients may permit internal exchanges of different types.

$$n : Amb[T], P : T \;\Rightarrow\; n[P] \text{ is legal and } n[P] : S$$

- If *n* permits *T*-exchanges, then *open n* may unleash *T*-exchanges in the current location.

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

- Any process that uses a *Cap*[*T*] had better be a process that already exchanges *T*, because new *T*-exchanges may be unleashed.

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

Judgments

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

Rules

(Env ø)

$$\frac{}{\emptyset \vdash \lozenge}$$

(Env *n*)

$$\frac{E \vdash \lozenge \qquad n \notin dom(E)}{E, n{:}W \vdash \lozenge}$$

(Exp *n*)

$$\frac{E', n{:}W, E'' \vdash \lozenge}{E', n{:}W, E'' \vdash n : W}$$

(Exp ε)

$$\frac{E \vdash \lozenge}{E \vdash \varepsilon : Cap[T]}$$

(Exp .)

$$\frac{E \vdash M : Cap[T] \qquad E \vdash M' : Cap[T]}{E \vdash M.M' : Cap[T]}$$

(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)

$$\frac{E \vdash M : Cap[T] \qquad E \vdash P : T}{E \vdash M.P : T}$$

(Proc Amb)

$$\frac{E \vdash M : Amb[T] \qquad E \vdash P : T}{E \vdash M[P] : S}$$

(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)

$$\frac{E \vdash M_1{:}W_1 \quad \ldots \quad E \vdash M_k{:}W_k}{E \vdash \langle M_1, \ldots, M_k \rangle : W_1 \times \ldots \times W_k}$$

(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)

$$\frac{E \vdash P : T \qquad E \vdash Q : T}{E \vdash P \mid Q : T}$$

(Proc Repl)

$$\frac{E \vdash P : T}{E \vdash \mathbf{!}P : T}$$

- Ex.: A capability that may unleash *S*-exchanges:
  $n{:}Amb[T], m{:}Amb[S] \vdash in\ n.\ open\ m : Cap[S]$
- Ex.: A process that outputs names of quiet ambients:
  $E \vdash \mathbf{!}(\nu n{:}Amb)\langle n\rangle : Amb$

- Proposition (Subject Reduction)
  If $E \vdash P : T$ and $P \longrightarrow Q$ then $E \vdash Q : T$.

Exercise

- Construct a typing derivation for the message example:

$(\nu\ a{:}\ Amb[Shh])$
$(\nu\ b{:}\ Amb[Amb[Shh]])$
$(\nu\ msg{:}\ Amb[Amb[Shh]])$
$\quad a[msg[\langle M\rangle \mid out\ a.\ in\ b]] \mid$
$\quad b[open\ msg.\ (n{:}Amb[Shh]).\ P]$

Typed Polyadic Asynchronous π-calculus

$$[\![E \vdash P]\!] \triangleq [\![E]\!] \vdash [\![P]\!] : Shh$$

$$[\![\varnothing, n_1{:}W_1, \ldots, n_k{:}W_k]\!] \triangleq \varnothing, n_1{:}[\![W_1]\!], n^p{}_1{:}[\![W_1]\!], \ldots, n_k{:}[\![W_k]\!], n^p{}_k{:}[\![W_k]\!]$$

$$[\![Ch[W_1, \ldots, W_k]]\!] \triangleq Amb[[\![W_1]\!]\times[\![W_1]\!]\times\ldots\times[\![W_k]\!]\times[\![W_k]\!]]$$

create the *n* buffer

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

create an *n* packet

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

enter the *n* buffer

$$[\![n(n_1{:}W_1, \ldots, n_k{:}W_k).P]\!] \triangleq$$

$$(\nu q{:}Amb[Shh])\ (open\ q \mid$$

$$n^p[in\ n.\ (n_1,n^p{}_1{:}[\![W_1]\!], \ldots, n_k,n^p{}_k{:}[\![W_{\mathrm{k}}]\!]).\ q[out\ n.\ [\![P]\!]]])$$

climb out of the *n* buffer

$$[\![P \mid Q]\!] \triangleq [\![P]\!] \mid [\![Q]\!]$$

$$[\![!P]\!] \triangleq\ ![\![P]\!]$$

- The previous encoding emulates the π-calculus, but:
  - Channel buffers are generated at ν occurrences.
  - If freely embedded within ambients, channel I/O may then fail if the channel buffer is not where the I/O happens, even if I and O are in the same place. (I.e., extrusion across ambient boundaries is not implemented by this encoding.)
  - Using 2 ambient names for 1 π name is a bit awkward.
- Georges Gonthier devised a different encoding:
  - Uses 1 ambient name for 1 π name.
  - New buffers are generated whenever needed to do I/O.
  - Encoding can be freely merged with ambient operations (I's and O's on a channel *n* interact when they are in the same ambient.)
  - Buffers must be *coalesced* to allow I/O interactions.

Gonthier's Coalescing Encoding

$$[\![Ch[W_1, \ldots, W_k]]\!] \triangleq Amb[[\![W_1]\!] \times \ldots \times [\![W_k]\!]]$$

create no buffers

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

create an *n* buffer

open any *n* buffer that enters

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

enter any *n* buffer

climb out of
coalescing towers
of *n* buffers

$$[\![n(n_1{:}W_1, \ldots, n_k{:}W_k).P]\!] \triangleq (\nu q{:}Amb[Shh])\ (open\ q.q[\,] \mid n[!open\ n \mid in\ n \mid (n_1{:}[\![W_1]\!], \ldots, n_k{:}[\![W_\mathrm{k}]\!]).\ q[!out\ n \mid open\ q.\ [\![P]\!]]])$$

$$[\![P \mid Q]\!] \triangleq [\![P]\!] \mid [\![Q]\!]$$

$$[\![!P]\!] \triangleq ![\![P]\!]$$

Typed Call-by-Value λ-calculus

$$[\![E \vdash b:T]\!] \triangleq [\![E]\!] \vdash (\nu^{\pi} k:Ch[[\![T]\!]])\ [\![b]\!]_k : Shh$$

$$[\![A \to B]\!] \triangleq Ch[[\![A]\!], Ch[[\![B]\!]]]$$

$$[\![x_T]\!]_k \triangleq k\langle x\rangle$$

$$[\![\lambda x:A.b_{A\to B}]\!]_k \triangleq$$
$$(\nu^{\pi} n:[\![A\to B]\!])\ (k\langle n\rangle \mid {!n}(x:[\![A]\!], k':Ch[[\![B]\!]]).\ [\![b_B]\!]_{k'})$$

$$[\![b_{A\to B}(a_A)]\!]_k \triangleq$$
$$(\nu^{\pi} k':Ch[[\![A\to B]\!]], k'':Ch[[\![A]\!]])$$
$$([\![b]\!]_{k'} \mid k'(x:[\![A\to B]\!]).\ ([\![a]\!]_{k''} \mid k''(y:[\![A]\!]).\ x\langle y, k\rangle))$$

$$[\![x:T \vdash x:T]\!]$$
$$= [\![x:T]\!] \vdash (\nu^{\pi} k:Ch[[\![T]\!]])\ k\langle x\rangle : Shh$$
$$= x:[\![T]\!], x^p:[\![T]\!] \vdash (\nu k:Amb[[\![T]\!]\times[\![T]\!]])\ k[!open\ k^p] \mid k^p[in\ k.\langle x, x^p\rangle] : Shh$$

Generalizations

- The *Amb*-*Cap* style of types and rules is very robust and extensible to many situations.
  - It works for all kinds of *effects* (not just exchanges).
  - *Amb* types for names.
  - *Cap* types for capabilities (to deal with *open*).
- Sketch of possible extensions:
  - Instead of a single type $Amb[T]$ for all ambient names that allow T exchanges, we can allow types $G[T]$, for distinct groups G from a fixed collection. (Akin to Milner's sort system for π).
  - Further, we can allow a subgroup hierarchy $G' <: G$, with *Amb* as the top group, inducing a subtype hierarchy.
  - Further, we can allow the creation of new groups G, as in $(\nu G)(\nu n{:}G)\ P$ or $(\nu G' <: G)(\nu n{:}G')\ P$.

Types for Mobility Control

- Effects
  - An effect is anything a process can do that we may want to track.
  - Then, $E \vdash P : F$ is interpreted to mean that P *may have at most effects* F. Works well for composition.
  - And $Amb[F]$ is an ambient that allows at most effects F.
  - And $Cap[F]$ is a capability that can unleash at most effects F.
- Applications
  - We have seen the case where an effect is an input or output operation of a certain type.
  - We can also consider *in* and *out* operations as effects. We can then use a type system to statically prevent certain movements.
  - We can also consider *open* operations as effects. We can then use a type system to statically prevent such operations.
  - To do all this without dependent types, we use groups.

Name Groups

- Name Groups have a variety of uses:
  - We would like to say, within a type system, something like:

    The ambient named *n* can enter the ambient named *m*.

    But this would bring us straight into *dependent types*, since names are value-level entities. This is *no fun at all*.
  - Instead, we introduce type-level name groups *G*,*H*, and we say:

    Ambients of group *G* can enter ambients of group *H*.
  - Groups are akin to π-calculus sorting mechanisms. We call them groups in the Unix sense of collections of principals.

Crossing Control

G, H	groups
$Hs ::= \{H_1..H_k\}$	sets of groups
$W ::=$	message types
$G[\frown Hs,T]$	ambient name in group G, containing processes that may cross Hs and exchange T
$Cap[\frown Hs,T]$	capability unleashing Hs crossings and T exchanges

$E \vdash P : \frown Hs,T$	process that exchanges T and crosses Hs
$\nu n{:}G[\frown\{\,\},T]$	a name for immobile ambients

Opening Control

$W ::=$	message types
$G[{}^\circ Hs,T]$	ambient name in group G, containing processes that may open Hs and exchange T
$Cap[{}^\circ Hs,T]$	capability unleashing Hs openings and T exchanges

$E \vdash P : {}^\circ Hs,T$	process that exchanges T and opens Hs
$\nu n{:}G[{}^\circ\{\},T]$	a name for locked ambients (where $G \notin \{\}$)

(Here n cannot be opened, because we require $G \in Hs$ for *open* n to be typeable, when $n{:}G[{}^\circ Hs,T]$. This is because the opening of G may unleash further openings of Hs. With this rule the transitive closure of possible openings must be present already in the given types. It also makes n above unopenable.)

Types for Secrecy Control

- In addition to static groups, we add ***group creation***.
  - This is a new construct for generating type-level names.
  - It can be studied already in π-calculus:

    $(\nu G)(\nu x{:}G)(\nu y{:}G)\ldots$
    Create a new group (collection of names) G
    and populate it with new elements x and y

    - Simply by type-checking, we can guarantee that a fresh x cannot escape the scope of G.
    - It can statically block certain communications that would be allowed by scope extrusion.
    - We can therefore prevent the "accidental" escape of capabilities that is a major concern in practical systems.
  - In ambient calculus, it further allows the safe sharing of secret between mobile processes.

Making Secrets

- Consider a player P and an opponent O:

  $O \mid P$

- In the π-calculus, if P is to create a fresh secret not shared with O, we program it to evolve into:

  $O \mid (\nu x)P'$

- **Name creation** $(\nu x)P'$ makes a fresh name x, whose scope is the process P'

Leaking Secrets

- Now, if the system were to evolve into this, the privacy of x would be violated:

  $p(y).O' \mid (\nu x)(p\langle x\rangle \mid P'')$

  (Output $p\langle x\rangle$ may be accidental or malicious.)

- By extrusion, this is $(\nu x)(\, p(y).O' \mid p\langle x\rangle \mid P'')$ which evolves to $(\nu x)(O'\{y \leftarrow x\} \mid P'')$
- So, the secret x has leaked to the opponent.

Trying to Prevent Leakage

- How might we prevent leakage?
  - Restrict output: not easy to prevent $p\langle x\rangle$ as p may have arisen dynamically
  - Restrict extrusion: again difficult, as it's needed for legitimate communication
- Can we exploit a sorted π-calculus?
  - Declare x to be of sort *Private*. But sorts are global, so the opponent can be type-checked.

$$p(y{:}Private).O' \mid (\nu x{:}Private)(p\langle x\rangle \mid P'')$$

Group Creation

- We want to be able to create fresh groups (sorts) on demand, and to create fresh elements of these groups on demand.
- We extend the sorted π-calculus with group creation $(\nu G)P$, which makes a new group G with scope P.
- Group creation obeys scope extrusion laws analogous to those for name creation.

Preventing Leakage

- We can now prevent leakage to a well-typed opponent by type-checking and lexical scoping (where $G[]$ is the type of nullary channels of group G):

$$p(y{:}T).O' \mid (\nu G)(\nu x{:}G[])(p\langle x\rangle \mid P'')$$

- The opponent $p(y{:}T).O'$ cannot be typed: the type T would need to mention G, but G is out of scope.

Untyped Opponents

- We cannot realistically expect the opponent to be well-typed.
- Can an untyped opponent, by cheating about the type of the channel p, somehow acquire the secret x?
- No, provided the player is typed; in particular, provided $p\langle x\rangle$ is typed.

Secrecy

- A player creating a fresh G cannot export elements of G outside the initial scope of G,
  - either because a well-typed opponent cannot name G to receive a message,
  - or because a well-typed player cannot use public channels to transmit G elements.
- In sum: channels of group G remain **secret**, forever, outside the initial scope of (νG).

Summary

- We have reduced secrecy of names to scoping and typing; subtleties include:
    - extrusion rules associated with scoping
    - leakage allowed by name extrusion
    - the possibility of untyped opponents
- A reasonable precondition of our results is that the player (but not the opponent) be type-checked in some global environment.

Secrecy in Typed Contexts

- For well-typed opponents, subject reduction alone has secrecy implications.

Theorem (Subject Reduction)
If $E \vdash P$ and $P \equiv Q$ then $E \vdash Q$.
If $E \vdash P$ and $P \rightarrow Q$ then $E \vdash Q$.

Corollary (No Leakage)
Let $P = p(y{:}T).O' \mid (\nu G)(\nu x{:}G[T])P'$. If $E \vdash P$ for some E then there are no Q', Q'', $C\{-\}$ such that $P \equiv (\nu G)(\nu x{:}G[T])Q'$ and $Q' \rightarrow Q''$ and $Q'' \equiv C\{p\square x\square\}$ where p and x are not bound by $C\{-\}$.

Secrecy in Untyped Contexts

Theorem (Secrecy)
Suppose that $(\nu G)(\nu x{:}T)P$ where G free in T. Let S be the names occurring in $dom(E)$. Then the type erasure $(\nu x)erase(P)$ of $(\nu G)(\nu x{:}T)P$ preserves the secrecy of the restricted name x from S.

Where "preserves the secrecy" is defined (in the paper) in terms of interactions with an opponent idealized as a set of names. It is similar to Abadi's definition for spi.

Instances and Applications

- There seems to be a link between group creation and several unusual type systems:
  - *letregion* in Tofte and Talpin's region analysis
  - *newlock* in Flanagan and Abadi's lock types
  - *runST* in Launchbury and Peyton Jones' lazy functional state threads
- Elsewhere, Dal Zilio and Gordon formalize the link with regions, and Cardelli, Ghelli and Gordon apply (νG) to regulate mobility.

Typed Ambient Calculus with Group Creation

- Start with exchange types.
- Just one new process construct:

  $$(\nu G)P$$

  to create a new group G with scope P.
- Just one modified type construct:

  $$G[T]$$

  as the type of names of group G that name ambients that contain T exchanges.
- The construct $G[T]$ replaces $Amb[T]$, where Amb can now be seen as the group of all names. So we can now write:

$$(\nu G)\ (\nu n{:}G[Int])\ n[\langle 3\rangle \mid (x{:}Int).\ P]$$

Types

$W ::=$	message types
$G[T]$	ambient name in group G with T exchanges
$Cap[T]$	capability unleashing T exchanges
$T ::=$	process types
Shh	no exchange
$W_1 \times \ldots \times W_k$	tuple exchange ($\mathbf{1}$ is the null product)

- A quiet ambient: $G[Shh]$
- A harmless capability: $Cap[Shh]$
- A synchronization ambient: $G[\mathbf{1}]$
- Ambient containing harmless capabilities: $G[Cap[Shh]]$
- A capability that may unleash the exchange of names for quiet ambients: $Cap[G[Shh]]$

Processes and Messages

$P, Q ::=$

$(\nu G)P$

new group

$(\nu n{:}W)P$

0

$P \mid Q$

$!P$

$M[P]$

$M.P$

$(n_1{:}W_1, \ldots, n_k{:}W_k).P$

$\langle M_1, \ldots, M_k \rangle$

$M, N ::=$

n

$in\ M$

$out\ M$

$open\ M$

ε

$M.N$

νG is static: type rules handle such G's.

νG is dynamic/generative: $!(\nu G)P$ not the same as $(\nu G)!P$.

Reduction

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

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

$$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$$

$$(n_1{:}W_1, \dots, n_k{:}W_k).P \mid \langle M_1, \dots, M_k \rangle \longrightarrow P\{n_1 \leftarrow M_1, \dots, n_k \leftarrow M_k\}$$

new group

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

$$P \longrightarrow Q \quad \Rightarrow \quad (\nu n{:}W)P \longrightarrow (\nu n{:}W)Q$$

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

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

$$P' \equiv P,\ P \longrightarrow Q,\ Q \equiv Q' \quad \Rightarrow \quad P' \longrightarrow Q'$$

Structural Congruence

- A usual.
  - $(\nu G)P$ is similar to $(\nu n{:}W)P$, including scope extrusion.

$$P \equiv Q \quad \Rightarrow \quad (\nu G)P \equiv (\nu G)Q$$

$$(\nu G)(\nu G')P \equiv (\nu G')(\nu G)P$$

$$(\nu G)(\nu n{:}W)P \equiv (\nu n{:}W)(\nu G)P \quad \text{if } G \notin fg(W)$$

$$(\nu G)(P \mid Q) \equiv P \mid (\nu G)Q \quad \text{if } G \notin fg(P)$$

$$(\nu G)(m[P]) \equiv m[(\nu G)P]$$

$$(\nu G)\mathbf{0} \equiv \mathbf{0}$$

- Extrusion of (νG) allows ambients to establish shared secrets, then go arbitrarily far away, and then come back to share the secrets. Without been able to give them away.

Judgments

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

Rules

(Env ø)

$$\frac{}{\varnothing \vdash \diamond}$$

(Env *n*)

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

(Env *G*)

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

(Type *G*)

$$\frac{G \in dom(E) \qquad E \vdash T}{E \vdash G[T]}$$

(Type *Cap*)

$$\frac{E \vdash T}{E \vdash Cap[T]}$$

(Type *Shh*)

$$\frac{E \vdash T}{E \vdash Shh}$$

(Type Tuple)

$$\frac{E \vdash W_1 \quad \ldots \quad E \vdash W_k}{E \vdash W_1 \times \ldots \times W_k}$$

(Exp *n*)

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

(Exp ε)

$$\frac{E \vdash Cap[T]}{E \vdash \varepsilon : Cap[T]}$$

(Exp .)

$$\frac{E \vdash M : Cap[T] \qquad E \vdash M' : Cap[T]}{E \vdash M.M' : Cap[T]}$$

(Exp In)

$$\frac{E \vdash n : G[S] \qquad E \vdash T}{E \vdash in\ n : Cap[T]}$$

(Exp Out)

$$\frac{E \vdash n : G[S] \qquad E \vdash T}{E \vdash out\ n : Cap[T]}$$

(Exp Open)

$$\frac{E \vdash n : G[T]}{E \vdash open\ n : Cap[T]}$$

(Proc Action)

$$\frac{E \vdash M : Cap[T] \qquad E \vdash P : T}{E \vdash M.P : T}$$

(Proc Amb)

$$\frac{E \vdash M : G[S] \qquad E \vdash P : S \qquad E \vdash T}{E \vdash M[P] : T}$$

(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)

$$\frac{E \vdash M_1{:}W_1 \quad \ldots \quad E \vdash M_k{:}W_k}{E \vdash \langle M_1, \ldots, M_k\rangle : W_1\times\ldots\times W_k}$$

(Proc GRes)

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

(Proc Res)

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

(Proc Zero)

$$\frac{E \vdash \diamond}{E \vdash \mathbf{0} : T}$$

(Proc Par)

$$\frac{E \vdash P : T \qquad E \vdash Q : T}{E \vdash P \mid Q : T}$$

(Proc Repl)

$$\frac{E \vdash P : T}{E \vdash\, !P : T}$$

- Prop (Subject Reduction)
  If $E \vdash P : T$ and $P \longrightarrow Q$
  then there exists Gs such that $Gs, E \vdash Q : T$.

Conclusions

- A new programming construct for expressing secrecy intentions.
- Good for "pure names" like channels, heap references, nonces, keys.
- Groups are like sorts, but no "new sort" construct has previously been studied.
- Basic idea could be added to any language, and is easily checked statically (no flow analysis…).