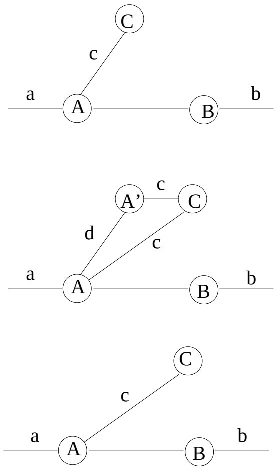
## Introduction to $\pi$-Calculus

- Motivation:
  how to describe mobile processes
- Basic idea: mobility as change of links
- Extending CCS with name passing to model mobility
- Reactions
- Equivalences
- Expressive power of $\pi$-calculus



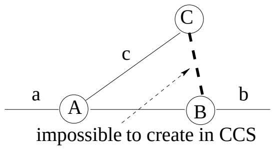
## (Im)Mobility



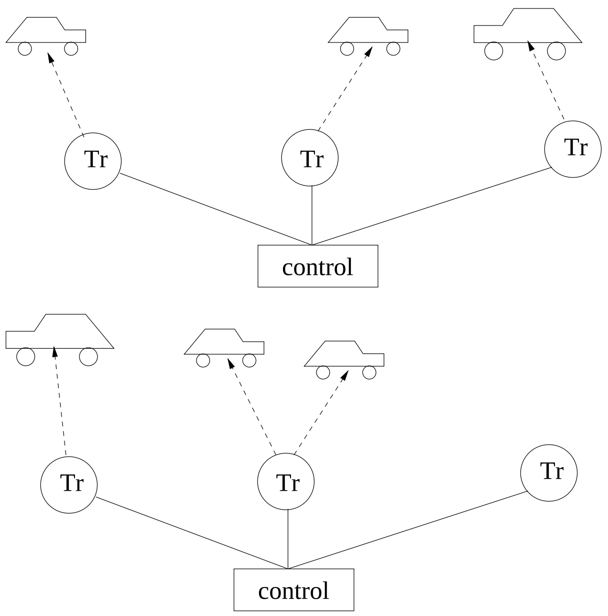

S = new c(A | C) | B

A = a.new d(A | A’) + c.A’’

A’ = c.0








## $\pi$-Calculus for Mobility

The view adopted:
Mobility = Creating **links** in a virtual space

Observation on CCS Process Expresions:
Name/coname shared by concurrent processes can be seen as a link.

The proposed solution:
Extend CCS Expressions by **name passing** mechanism. Passed names can create new links.

## Link creation by name passing

Draw flow graphs for (_|_|_):

$A = a(x).b'(x).c'(x).A$
$B = b(z).z(y).B$
$C = c(v).v'(d).C$

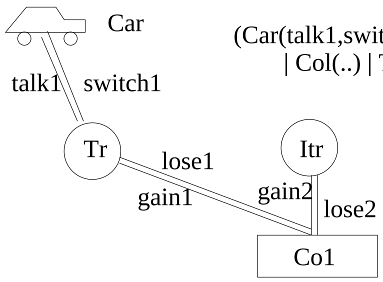
External reaction of $A$ with $a'(t)$
$A' = b'(t).c'(t).A$
$B = b(z).z(y).B$
$C = c(v).v'(d).C$

Reaction $A'$ - $B$
$A'' = c'(t).A$
$B' = t(y).B$
$C = c(v).v'(d).C$

Reaction $A''$ - $C$
$A = a(x).b'(x).c'(x).A$
$B' = t(y).B$
$C' = t'(d).C$



## $\pi$-Calculus Example




(Car(talk1,switch1) |
| Col(..) | Tr(..) | Itr)

Car(talk,switch) = 'talk.Car(talk,switch)+switch(t,s).Car(t,s)

Co1 = 'lose1(talk2,switch2).'gain2(talk2,switch2).Co2

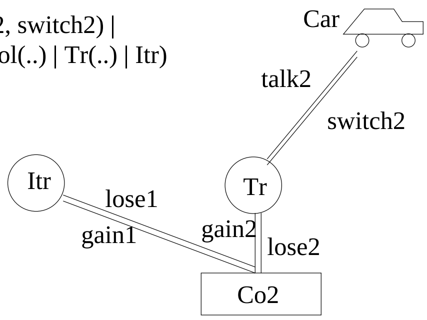
Co2 = 'lose2(talk1,switch1).'gain1(talk1,switch1).Co1

Tr(talk,switch,gain,lose) = talk.Tr(talk,switch,gain.lose)+
lose(t,s).'switch(t,s).Itr(gain,lose)

Itr(gain,lose) = gain(t,s).Tr(t,s,gain,lose)



(Car(talk2, switch2) |
| Col(..) | Tr(..) | Itr)




Car(talk,switch) = 'talk.Car(talk,switch)+switch(t,s).Car(t,s)

Co1 = 'lose1(talk2,switch2).'gain2(talk2,switch2).Co2

Co2 = 'lose2(talk1,switch1).'gain1(talk1,switch1).Co1

Tr(talk,switch,gain,lose) = talk.Tr(talk,switch,gain.lose)+
lose(t,s).'switch(t,s).Itr(gain,lose)

Itr(gain,lose) = gain(t,s).Tr(t,s,gain,lose)



## Monadic $\pi$-calculus

- Action prefixes $\pi$: **one-argument** names
  - $x(y)$ receive $y$ along $x$,
  - $'x(y)$ send $y$ along $x$,
  - $\tau$ unobservable action

$$P ::= \Sigma \pi_i . P_i$$
$$(P_1 | P_2)$$
$$new\ a\ P$$
$$!P$$

0 - the empty sum, no action
$!P$ - *replication*:
$!P \equiv P|!P$

Recursive definitions are not needed

## The structural congruence $\equiv$

**Intuition:** $\equiv$ describes simplest cases of semantic equivalences (semantically equivalent syntactic variants of expressions )

- Change of bound names
  $new\ aP(a) \equiv new\ bP(b)$
- reordering in summation
  $P + Q \equiv Q + P$
- Concurrency operator:
  $P|0 \equiv P,\ P|Q \equiv Q|P,$
  $P|(Q|R) \equiv (P|Q)|R$
- Hiding:
  $new\ x\ (P|Q) \equiv P|new\ x\ Q$ if $a \notin fn(P)$
  $new\ a\ 0 \equiv 0,\ new\ ab\ P \equiv new\ ba\ P$
- Replication:
  $P! \equiv P|!P$



## Definitions replaced by Replications

Consider a definition:

$$A(x) = Q$$

e.g. $A(x) = x(y).A(x)$
Assume $P$ is the scope of this definition,
e.g. $(A(q)|0)$

- choose a new name $a$
- transform every call $A(t)$ in $P$ into $'a(t)$, denote the result $\widehat{P}$
- Replace $P$ by $(new\ a\ (\widehat{P}|!a(x).\widehat{Q}))$

$$new\ a\ ('a(q)|!p(x).x(y).a(x))$$

How it works:
$('a(q)|0|!a(x).x(y).'a(x))$
$\equiv$
$('a(q)|a(x).x(y).'a(x)|!a(x).x(y).'a(x))$
$\rightarrow$
$(q(y).'a(q)|!a(x).x(y).'a(x))$



## Reaction Rules

$$TAU\ :\ \tau.P + M \rightarrow P$$

$$REACT\ :\ (x(y).P + M)|('x(z).Q + N) \rightarrow \{z/y\}P|Q$$

$$PAR\ :\ \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$RES\ :\ \frac{P \rightarrow P'}{new\ x\ P \rightarrow new\ x\ P'}$$

$$STR\ :\ \frac{P \rightarrow P'}{Q \rightarrow Q'}\ if\ P \equiv Q\ and\ P' \equiv Q'$$

$\{z/y\}P$ : replace all **free occurrences** of $y$ in $P$ by $z$
$x$ can be bound by hiding $new\ x$ and
by input action $y(x)$ (but not by $'y(x)$)

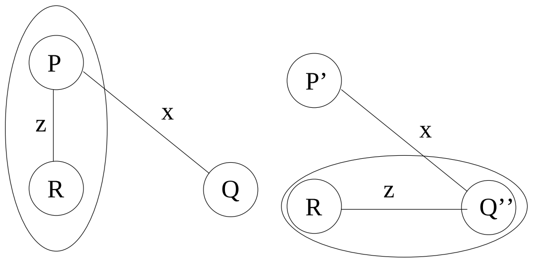


## Example reduction

$x(z).'y(z)|!(new\ y('x(y).Q))$
$\equiv$
$x(z).'y(z)|!(new\ v('x(v).Q))$
$\equiv$
$x(z).'y(z)|(new\ v('x(v).Q))|!(new\ v('x(v).Q))$
$\equiv$
$(new\ v(x(z).'y(z)|'x(v).Q))|!(new\ v('x(v).Q))$
$\rightarrow$
$(new\ v('y(v)|Q)|!(new\ v('x(v).Q))$

## Mobility

$P =' x(z).P'$
$Q = x(y).Q'$
$z$ free in $P, R$,
$x$ free in $P, Q$
$z$ not free in $P'$ nor in $Q$
$(new\ z(P|R)|Q) \equiv (new\ z('x(z).P'|R)|x(y).Q') \equiv$
$(new\ z('x(z).P'|R|x(y).Q')) \rightarrow (new\ z(P'|R|\{z/y\}Q')) \equiv$
$(P'|new\ z(R|\{z/y\}Q'))$






## Polyadic $\pi$-Calculus

- messages may include more tham one name
- this can be encoded in the monadic calculus

$x(y_1, ..., y_n).P|'x(z_1, ..., z_n).Q$

Encoding idea:

- create and send *new* name
- use it to transfer the components one-by-one

$x(y_1, ..., y_n).P \rightsquigarrow x(w).w(y_1)...w(y_n).P$

$'x(z_1, ..., z_n).Q \rightsquigarrow new\ w\ ('x(w).'w(z_1)...'w(z_n).Q)$



## Abstractions and Concretions

Motivation: to identify basic operations of $\pi$-calculus.

- observable actions, like CCS
- reactions, like CCS
- application **new!**
  (like parameter passing in $\lambda$-calculus ?)

$n$-ary **abstraction**:

$$(a_1, ..., a_n).P$$

$n$-ary **concretion** ($\vec{z} \subseteq \vec{y}$):

$$new\ \vec{z}\ \langle\vec{y}\rangle.Q$$

**reaction** (on abstraction $F$ and concr. $C$):

$$(xF + M)|('xC + N) \rightarrow F@C$$

**application**

$$(\vec{x}).P@new\ \vec{z}\langle\vec{y}\rangle.Q = new\ \vec{z}(\{\vec{y}/\vec{x}\}P|Q)$$

**agent**: arbitrary abstraction or concretion (hence also any process).



## Commitment Rules

$$SUM\ :\ M + \alpha A + N \rightarrow^{\alpha} A$$

$$L-R\ :\ \frac{P \rightarrow^{x} F \quad Q \rightarrow^{'x} C}{P|Q \rightarrow F@C}$$

$$R-R\ :\ \frac{P \rightarrow^{'x} C \quad Q \rightarrow^{x} F}{P|Q \rightarrow F@C}$$

$$L-P\ :\ \frac{P \rightarrow^{\alpha} A}{P|Q \rightarrow^{\alpha} A|Q}$$

$$R-P\ :\ \frac{Q \rightarrow^{\alpha} A}{P|Q \rightarrow^{\alpha} A|P}$$

$$RES\ :\ \frac{P \rightarrow^{\alpha} A}{new\ x\ P \rightarrow^{\alpha} new\ x\ A}\quad if\ \alpha \notin \{x,' x\}$$

$$REP\ :\ \frac{P|!P \rightarrow^{\alpha} A}{!P \rightarrow^{\alpha} A}$$

## Strong Equivalence of Agents

**Strong simulation**: whenever $P\mathcal{S}Q$:
if $P \to^{\alpha} A$ then $Q \to^{\alpha} B$
for some $B$ and $A\mathcal{S}B$

**But**: $A, B$ are agents, not processes.

Extending $\mathcal{S}$

- for **abstractions** $F = (x).P, G = (x).Q$
  $F\mathcal{S}G$ means :
  for all $y$ $F\langle y\rangle \mathcal{S} G\langle y\rangle$
  **thus** $\{y/x\}P\mathcal{S}\{y/x\}Q$ for **all** $y$
- for **concretions**
  $C \equiv new\ z\langle y\rangle.P$ and $D \equiv new\ z\langle y\rangle.Q$
  $C\mathcal{S}D$ means $P\mathcal{S}Q$

**Strong equivalence** defined by strong bisimulation



## Agent congruence

Substitution may destroy $\sim$

$$'x|y \sim\ 'x.y + y.'x \quad if\ x \neq y$$

Substitute $\{x/y\}$

$$'x|x \not\sim\ 'x.x + x.'x$$

Why?

**Agent congruence**:
an equivalence $\cong$ on agents such that:

- If $A \cong B$ then $\alpha A + m \cong \alpha B + m$ ($M$ any sum)
- If $P \cong Q$ then
  - $new\ aP \cong new\ aQ$
  - $P|R \cong Q|R$, and $R|P \cong R|Q$
  - $!P \cong !Q$
  - (concretions) $new\ x\langle y\rangle.P \cong new\ x\langle y\rangle.Q$
- (**abstractions**) if $\{y/x\}P \cong \{y/x\}Q$ for all $y$ then $(x).P \cong (x).Q$

$\sim$ is an agent congruence.



## Expressive Power of $\pi$-calculus

Can simulate $\lambda$-calculus

- Translation:
  any $\lambda$-term $M$ to a $\pi$-expression $[[M]]$ s.that

$$[[(\lambda x.M)N]] \approx [[\{N/x\}M]]$$

The translation:

- $[[x]](u) =\ 'x\langle u\rangle$
- $[[\lambda x.M]](u) = u(xv).[[M]]\langle v\rangle$
- $[[(M\ N)]](u) = new\ v([[M]]\langle v\rangle|new\ x('v\langle xu\rangle|!x[[N]]))$
  where chosen $x$ is not free in $N$

$$(\lambda x.x)N \to N$$

$[[(\lambda x.x)]]\langle v\rangle = v(xw).'x\langle w\rangle$
$[[(\lambda x.x)N]]\langle u\rangle =$
$new\ v(v(xw).'x\langle w\rangle|new\ y('v(yu)|!y[[N]]))$ thus

$$[[(\lambda x.x)N]]]\langle u\rangle \to^* [[N]]\langle u\rangle|new\ y!y[[N]]$$

But

$$new\ y!y[[N]] \sim 0$$

Hence

$$[[(\lambda x.x)N]]\langle u\rangle \to^* \sim [[N]]$$



## Summary $\pi$-calculus

- Motivated by mobile processes
- Describes mobility as linking
- Extends CCS by name passing
- Concepts/Equivalences/Results of CCS extend naturally to $\pi$-calculus
- Can be used as foundation for computability theory