Course on Mobility

Daniel.Hirschkoef@ens-lyon.fr
About this course

- focus on the $\pi$-calculus: a calculus of mobile processes based on naming (cf. R. Milner, Turing award lecture)
About this course

- focus on the π-calculus: a calculus of mobile processes based on naming (cf. R. Milner, Turing award lecture)

- π as a specification programming language
About this course

- focus on the π-calculus: a calculus of mobile processes based on naming (cf. R. Milner, Turing award lecture)
- π as a specification programming language
- more a panorama than a precise technical study of a particular point
About this course

• focus on the $\pi$-calculus: a calculus of mobile processes based on naming (cf. R. Milner, Turing award lecture)

• $\pi$ as a specification programming language

• more a panorama than a precise technical study of a particular point

• outline:
  $\pi$: definition - types - $\lambda$ in $\pi$ - behavioural equivalences
Origins and sources

- predecessors: other process algebras – CSP, CCS
Origins and sources

- predecessors: other process algebras – CSP, CCS

- books:
  R. Milner, *Communication and Concurrency*, Prentice Hall
  R. Milner, *Communicating and Mobile Systems: the π-calculus*, CUP
Origins and sources

- predecessors: other process algebras – CSP, CCS

- books:
  R. Milner, *Communication and Concurrency*, Prentice Hall
  R. Milner, *Communicating and Mobile Systems: the $\pi$-calculus*, CUP
  D. Sangiorgi, D. Walker, *The $\pi$-calculus, a Theory of Mobile Computation*, CUP

- notes for the course:
  not a tutorial, more to be used as a reference with the slides
Names and Processes

- nominal calculus: an infinite set of *names* *(channels, links, ports)*
  
  \[ a, b, \ldots, p, q, r, \ldots, x, y, \ldots \]

- we define *terms* *(processes)*
  
  \[ A, B, \ldots, P, Q, \ldots \]
Interaction, reduction, communication

\[ P = a\langle v \rangle . b(x).0 \mid a(y).(c\langle y \rangle.0 \mid d\langle y \rangle.0) \]
Interaction, reduction, communication

\[ P = a\langle v\rangle.b(x).0 \mid a(y).(c\langle y\rangle.0 \mid d\langle y\rangle.0) \]
\[ \downarrow \]
\[ b(x).0 \mid c\langle v\rangle.0 \mid d\langle v\rangle.0 \]
Interaction, reduction, communication

\[ P = \bar{a}\langle v \rangle . b(x) . 0 \mid a(y). (\bar{c}\langle y \rangle . 0 \mid \bar{d}\langle y \rangle . 0) \]
\[ \downarrow \]
\[ b(x) . 0 \mid \bar{c}\langle v \rangle . 0 \mid \bar{d}\langle v \rangle . 0 \]

competition for a resource:

\[ Q = a(x).Q_1 \mid a(x).Q_2 \mid \bar{a}\langle v \rangle . 0 \]
Interaction, reduction, communication

\[ P = \alpha \langle v \rangle . b(x).0 \mid a(y). (\bar{c} \langle y \rangle .0 \mid \bar{d} \langle y \rangle .0) \]

\[ \downarrow \]

\[ b(x).0 \mid \bar{c} \langle v \rangle .0 \mid \bar{d} \langle v \rangle .0 \]

competition for a resource:

\[ Q = a(x).Q_1 \mid a(x).Q_2 \mid \bar{a} \langle v \rangle .0 \]

\[ Q_{1\{x \leftarrow v\}} \mid a(x).Q_2 \mid 0 \]

\[ a(x).Q_1 \mid Q_{2\{x \leftarrow b\}} \mid 0 \]

non confluence
A single entity: names

- prefixes:

\[ a(b). \text{ reception}, \quad \bar{a}(b). \text{ emission} \quad \left\{ \begin{array}{l}
  a: \text{ subject} \\
  b: \text{ object}
\end{array} \right. \]
A single entity: names

- prefixes:
  \[ a(b). \text{reception}, \quad \overline{a}(b). \text{emission} \quad \begin{cases} a: \text{subject} \\ b: \text{object} \end{cases} \]

- communication:
  - synchronisation on a channel
  - substitution of a name with a name \((\not= \lambda)\)
A single entity: names

- prefixes:
  \[ a(b). \text{reception}, \quad \overline{a}(b). \text{emission} \]

- communication:
  - synchronisation on a channel
  - substitution of a name with a name \((\neq \lambda)\)

- often use names like \(x, y\) in input object (bound name)
A single entity: names

- prefixes:

  \[ a(b). \text{reception}, \quad \overline{a}\langle b \rangle. \text{emission} \]

\[
\begin{cases}
  a: \text{subject} \\
  b: \text{object}
\end{cases}
\]

- communication:
  - synchronisation on a channel
  - substitution of a name with a name (\( \neq \lambda \))

- often use names like \( x, y \) in input object (bound name)

- notation: \( \overline{a}\langle b \rangle.0 \) is often written \( \overline{a}\langle b \rangle \)
Another process

\( \overline{a} \langle c \rangle . \overline{c} \langle v \rangle . 0 \)
Another process

\[ \overline{a} \langle c \rangle . \overline{c} \langle v \rangle . 0 \mid a(x).x(t).\overline{r} \langle t \rangle . 0 \]
Another process

\[ \overline{a}\langle c \rangle.\overline{c}\langle v \rangle.0 \mid a(x).x(t).\overline{r}\langle t \rangle.0 \]

\[ \downarrow \]

\[ \overline{c}\langle v \rangle.0 \mid c(t).\overline{r}\langle t \rangle.0 \]
Another process

\[ \overline{a}\langle c\rangle.\overline{c}\langle v\rangle.0 \mid a(x).x(t).\overline{r}\langle t\rangle.0 \]
\[ \downarrow \]
\[ \overline{c}\langle v\rangle.0 \mid c(t).\overline{r}\langle t\rangle.0 \]
\[ \downarrow \]
\[ 0 \mid \overline{r}\langle v\rangle.0 \]
Another process

\[
\begin{array}{c}
\overline{a}(c).\overline{c}(v).0 \mid a(x).x(t).\overline{r}(t).0 \\
\downarrow
\end{array}
\]

\[
\begin{array}{c}
\overline{c}(v).0 \mid c(t).\overline{r}(t).0 \\
\downarrow
\end{array}
\]

\[
0 \mid \overline{r}(v).0
\]

- a form of *reference passing*

▷ object \leftrightarrow subject:  \( \overline{a}(c).\overline{c}(v), a(x).x(t).\overline{r}(t) \)
Another process

\[ \overline{a}(c).\overline{c}(v).0 \mid a(x).x(t).\overline{r}(t).0 \]
\[ \downarrow \]
\[ \overline{c}(v).0 \mid c(t).\overline{r}(t).0 \]
\[ \downarrow \]
\[ 0 \mid \overline{r}(v).0 \]

- a form of reference passing
- object \( \hookrightarrow \) subject: \( \overline{a}(c).\overline{c}(v) \), \( a(x).x(t).\overline{r}(t) \)
- name passing: the king of France, Google
Another process

\[ \overline{a}\langle c\rangle.\overline{c}\langle v\rangle.0 \mid a(x).x(t).\overline{r}\langle t\rangle.0 \]
\[ \Downarrow \]
\[ \overline{c}\langle v\rangle.0 \mid c(t).\overline{r}\langle t\rangle.0 \]
\[ \Downarrow \]
\[ 0 \mid \overline{r}\langle v\rangle.0 \]

- a form of reference passing
- object \(\rightarrow\) subject: \(\overline{a}\langle c\rangle.\overline{c}\langle v\rangle, a(x).x(t).\overline{r}\langle t\rangle\)
- name passing: the king of France, Google

- we have added a context: \(\overline{a}\langle c\rangle.\overline{c}\langle v\rangle.0\)
Another process

\[
\begin{align*}
\bar{a}\langle c \rangle.\bar{c}\langle v \rangle.0 & \mid a(x).x(t).\bar{r}\langle t \rangle.0 \\
\downarrow \quad & \\
\bar{c}\langle v \rangle.0 & \mid \underline{c}(t).\bar{r}\langle t \rangle.0 \\
\downarrow \quad & \\
0 & \mid \bar{r}\langle v \rangle.0
\end{align*}
\]

- a form of reference passing
  - object $\leftrightarrow$ subject: $\bar{a}\langle c \rangle.\bar{c}\langle v \rangle$, $a(x).x(t).\bar{r}\langle t \rangle$
  - name passing: the king of France, Google

- we have added a context: $\bar{a}\langle c \rangle.\bar{c}\langle v \rangle.0 \mid a(x).x(t).\bar{r}\langle t \rangle.0$
  - this is the way we reason on $\pi$-calculus terms
[\lambda \text{ versus } \pi]

\lambda: \text{ functions that are applied to their arguments (}\beta\text{-reduction)}
\pi: \text{ names being exchanged (}\simeq \beta_0\text{-reduction)}
λ versus π

λ: functions that are applied to their arguments (β-reduction)
π: names being exchanged (≃ β₀-reduction)

λ: a term being reduced, an evaluation that is going on
π: a term in a context
\[ \lambda \text{ versus } \pi \]

\(\lambda\): functions that are applied to their arguments (\(\beta\)-reduction)
\(\pi\): names being exchanged (\(\simeq \beta_0\)-reduction)

\(\lambda\): a term being reduced, an evaluation that is going on
\(\pi\): a term in a context

\(\lambda\): several kinds of reduction
  - strategies (call-by-name, call-by-value, \ldots)
  - computing everywhere in the term (rule \(\xi\))
\(\pi\): reduction only “at top-level”, non deterministically
Exercise: matching

- some \( \pi \)-calculi include a matching operator:
  \[ [n = m] P \] behaves like \( P \) if \( n = m \), is stuck otherwise

examples:
- \( a(x).b(y).[x = y] \overline{c}(x) \) forwards a name if received twice
- \( (\nu y) a(x).[x = y] P \) is equivalent to \( 0 \)
Exercise: matching

- some $\pi$-calculi include a matching operator:
  
  \[ [n = m] P \] behaves like $P$ if $n = m$, is stuck otherwise

  examples:
  - $a(x).b(y).[x = y] \overline{c}(x)$ forwards a name if received twice
  - $(\nu y) a(x).[x = y] P$ is equivalent to 0

- is matching encodable in a $\pi$-calculus without matching operator?
Restriction operator, $\nu$

$(\nu a)P$: the process $P$ in which name $a$ is private
(unknown to any other process, unknown to the context)
Restriction operator, $\nu$

$(\nu a) P$: the process $P$ in which name $a$ is private
(unknown to any other process, unknown to the context)
other interpretation: create a new name $a$, then execute $P$
Restriction operator, $\nu$

$(\nu a) P$: the process $P$ in which name $a$ is private

(unknown to any other process, unknown to the context)

Other interpretation: create a new name $a$, then execute $P$

Example:

\[ T = (\nu a) (\overline{a}\langle v \rangle \mid a(x).Q_1) \mid a(y).Q_2 \]

$\rightarrow$ no communication with “$Q_2$”
[Restriction operator, \(\nu\)]

\((\nu a) P\): the process \(P\) in which name \(a\) is **private**
(unknown to any other process, unknown to the context)
other interpretation: create a *new* name \(a\), then execute \(P\)

**Example:** \( T = (\nu a)(\overline{a}\langle v \rangle | a(x).Q_1) | a(y).Q_2 \)
\( \rightarrow \) no communication with “\(Q_2\)”

**Remarks:**
- \(\nu\) is a binder: \(T\) is \(\alpha\)-equivalent to
  \((\nu a')(\overline{a'}\langle v \rangle | a'(x).Q_1{_{a\leftarrow a'}}) | a(y).Q_2 \) \((a' \text{ fresh name})\)
Restriction operator, $\nu$

$(\nu a) P$: the process $P$ in which name $a$ is **private**
(unknown to any other process, unknown to the context)

other interpretation: create a *new* name $a$, then execute $P$

Example:  \[ T = (\nu a) (\overline{a}\langle v \rangle | a(x).Q_1) \mid a(y).Q_2 \]
\[ \rightarrow \text{no communication with "Q}_2" \]

Remarks:
- $\nu$ is a binder: $T$ is $\alpha$-equivalent to
  \[ (\nu a') (\overline{a'}\langle v \rangle | a'(x).Q_1\{a\leftarrow a'\}) \mid a(y).Q_2 \]
  \[ (a' \text{ fresh name}) \]
- $\nu$ has greater priority than $|$
Name extrusion

the object of an output is a restricted name

\[
(\nu c)(P \mid \bar{a}(c).Q) \mid a(x).R \rightarrow (\nu c)(P \mid Q \mid R_{\{x\leftarrow c\}}) \equiv (\nu c)(P \mid R_{\{x\leftarrow c\}}) \mid Q
\]

→ ‘network topology’ is changing along computation
Exercise: localised $\pi$

- grammar so far: $P ::= 0 \mid P_1 \mid P_2 \mid a(b).P \mid \overline{a}(b).P \mid (\nu n)P$
Exercise: localised $\pi$

- grammar so far: $P ::= 0 | P_1 | P_2 | a(b).P | \bar{a}\langle b \rangle.P | (\nu n)P$

- localised $\pi$: in $a(b).P$, $b$ can only be used in output

$\rightarrow$ why the name “localised $\pi$”? (consider a term of the form $(\nu n)P$)
The polyadic $\pi$-calculus

- possibility of exchanging *name tuples*:

\[
\bar{a}\langle u, v \rangle.P \parallel a(x, y).Q \rightarrow P \parallel Q\{x, y\leftarrow u, v\}
\]
The polyadic $\pi$-calculus

- possibility of exchanging *name tuples*:

$$\bar{a}\langle u, v \rangle . P \mid a(x, y) . Q \rightarrow P \mid Q\{x,y\leftarrow u,v\}$$

- remark: “type” errors

$$\bar{a}\langle u, v, w \rangle . P \mid a(x, y) . Q \rightarrow ??$$
The polyadic $\pi$-calculus

- possibility of exchanging *name tuples*:
  \[
  \overline{a}(u, v).P \mid a(x, y).Q \rightarrow P \mid Q_{\{x, y \leftarrow u, v\}}
  \]

- remark: “type” errors
  \[
  \overline{a}(u, v, w).P \mid a(x, y).Q \rightarrow \text{??}
  \]

- notation:
  \(a().P\) (resp. \(\overline{a}().P\)) is written \(a.P\) (resp. \(\overline{a}.P\)): cf. CCS
Booleans in the polyadic $\pi$-calculus

- an abstraction: $\text{true} \equiv (t, f).\bar{t}$

*cf. Milner's tutorial on $\pi$, abstractions and concretions*
Booleans in the polyadic $\pi$-calculus

- an abstraction: $\text{true} \overset{\text{def}}{=} (t, f).\bar{t}$

  cf. Milner's tutorial on $\pi$, abstractions and concretions

- the value true located at $b$: $\text{true}_b \overset{\text{def}}{=} b(t, f).\bar{t}$
Booleans in the polyadic $\pi$-calculus

- an abstraction: $\text{true} \overset{\text{def}}{=} (t, f).\overline{t}$
  
  cf. Milner's tutorial on $\pi$, abstractions and concretions

- the value true located at $b$: $\text{true}_b \overset{\text{def}}{=} b(t, f).\overline{t}$

- test:

  $\text{if } b \text{ then } P \text{ else } Q \overset{\text{def}}{=} \overline{b}(t, f).(t.P \mid f.Q)$
Booleans in the polyadic $\pi$-calculus

- an abstraction: $\text{true} \overset{\text{def}}{=} (t, f) \cdot \overline{t}$
  
  cf. Milner’s tutorial on $\pi$, abstractions and concretions

- the value true located at $b$: $\text{true}_b \overset{\text{def}}{=} b(t, f) \cdot \overline{t}$

- test:

  $$\text{if } b \text{ then } P \text{ else } Q \overset{\text{def}}{=} \overline{b}(t, f). (t.P \mid f.Q)$$

  $$\text{better} \overset{\text{def}}{=} (\nu t)(\nu f) \overline{b}(t, f).(t.P \mid f.Q)$$
Exercises

- write $\pi$-calculus terms for boolean $\neg$ and $\land$ operators
Exercises

• write $\pi$-calculus terms for boolean $\neg$ and $\land$ operators

• how can we ‘program’ the diadic $\pi$-calculus in the monadic $\pi$-calculus?

$$\overline{a}\langle u, v \rangle.P \mid a(x, y).Q \quad \rightarrow \quad P \mid Q\{x, y \leftarrow u, v\}$$
Replication

- to have a Turing-complete model (and in particular to be able to define a programming language), one has to have a form of recursion
[Replication]

- to have a Turing-complete model (and in particular to be able to define a programming language), one has to have a form of recursion

- replication: \[ !P \]

  stands for as many copies of \( P \) as you wish in parallel

\[ (!P \equiv P | P | P | \ldots) \]
Replication

- to have a Turing-complete model (and in particular to be able to define a programming language), one has to have a form of recursion

- replication: $!P$
  stands for as many copies of $P$ as you wish in parallel
  ($!P \equiv P | P | P | \ldots$)

- examples:
  $\overline{a}(v).P | !a(x).Q \rightsquigarrow P | Q_{\{x \leftarrow v\}} | !a(x).Q$
Replication

- to have a Turing-complete model (and in particular to be able to define a programming language), one has to have a form of recursion

- replication: $\text{!}P$

  stands for as many copies of $P$ as you wish in parallel

  ($\text{!}P \equiv P | P | P | ...$)

- examples:
  - $\text{a}(v).P | \text{!}a(x).Q \longrightarrow P | Q_{x\leftarrow v} | \text{!}a(x).Q$
  - let $T \overset{\text{def}}{=} \text{!c}(x) | \text{!c}(y)$, $T \longrightarrow T$

  → the replication operator brings persistence
Replication and persistence

- persistent data

\[ \text{true}_b \overset{\text{def}}{=} \neg b(t, f).\bar{t} \]
Replication and persistence

- persistent data
  \[ \text{true}_b \overset{\text{def}}{=} !b(t, f) . \bar{t} \]

- a resource: server for boolean \( \lor \)
  \[ !l(b_1, b_2, r). (\nu b) \left( !b(t, f). (\nu f') \left( \overline{b_1} \langle t, f' \rangle \mid f'. \overline{b_2} \langle t, f \rangle \right) \mid \overline{r} \langle b \rangle \right) \]
The language so far

\[ P ::= 0 \mid P_1 \mid P_2 \mid !P \mid a(b).P \mid \overline{a}\langle b\rangle.P \mid (\nu a)P \]

this \(\pi\)-calculus is:
- monadic
- synchronous
- with replication

but there exist several other variations/extensions