The full π-calculus: simple and expressive

π-Refresh. What we know so far?

- We have studied the asynchronous monadic π-calculus
  - no continuation on the output (asynchronous) $\overline{u}\langle v\rangle.P$
  - only one value is communicated (monadic) $\overline{u}\langle v\rangle$

$P, Q ::=$		processes
	$\mathbf{0}$	nil process
	$P \mid Q$	parallel composition of P and Q
	$(\nu\, a)P$	generation of a with scope P (also called *restriction*)
	$!P$	replication of P, i.e. infinite parallel composition $P \mid P \mid P \mid \ldots$
	$\overline{u}\langle v\rangle$	output of v on channel u
	$u(x).P$	input of *distinct* variables x on u, with continuation P

π-calculus: simple, but expressive

- Why expressive:
  - encoding data structures
  - encoding polyadic communication with monadic primitives
  - encoding synchronous communication with asynchronous (Honda/Tokoro, Boudol)
  - encoding choice (Nestmann, Palamidessi)
  - encoding recursion
  - encoding Higher order functions

Today you will learn how to master that expressiveness ... !!!

Synchronous π-calculus

- It is time to add continuation on output: $\overline{u}\langle v \rangle.P$

- We can define the synchronous calculus as follows:

$$\overline{a}\langle b \rangle.P \,|\, a(x).Q \longrightarrow P \,|\, Q\{^b/_x\}$$

Q: Can we simulate synchronous communication with asynchronous?
Hint: we need additional messages

Basic Encoding Definition

- $[\![P]\!] = Q$ is a function (mapping) from P to Q.

- A good mapping should be homomorphic.

– $[\![\mathbf{0}]\!] = \mathbf{0}$

– $[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$

– $[\![(\nu\, a)P]\!] = (\nu\, a)[\![P]\!]$

– $[\![!P]\!] = ![\![P]\!]$

- Question:

– $[\![\overline{a}\langle b\rangle.P]\!] =$ some asynchronous π-term

– $[\![a(x).P]\!] =$ some asynchronous π-term

Synchronous π-calculus

Synchronous $\qquad\qquad\qquad\qquad$ Asynchronous

$\Downarrow \qquad\qquad\qquad\qquad\qquad\qquad \Downarrow$

$$[\![\overline{u}\langle v\rangle.P]\!] = (\nu\, c)(\overline{u}\langle c\rangle \mid c(y).(\overline{y}\langle v\rangle \mid [\![P]\!])) \qquad \text{where } y \notin \mathit{fv}(P), c \notin \mathit{fn}(P)$$

$$[\![u(x).P]\!] = u(y).(\nu\, d)(\overline{y}\langle d\rangle \mid d(x).[\![P]\!]) \qquad \text{where } y \notin \mathit{fv}(P), d \notin \mathit{fn}(P)$$

- Note: $[\![P]\!]$ represents the formal notation for the encoding of P

- Example:

$$\begin{array}{rcl} [\![\overline{b}\langle e\rangle.P]\!] \mid [\![b(x).Q]\!] & \longrightarrow & (\nu\, c)(c(y).(\overline{y}\langle e\rangle \mid [\![P]\!]) \mid (\nu\, d)(\overline{c}\langle d\rangle \mid d(x).[\![Q]\!])) \\ & \longrightarrow & (\nu\, d)(\overline{d}\langle e\rangle \mid [\![P]\!] \mid d(x).[\![Q]\!]) \longrightarrow [\![P]\!] \mid [\![Q]\!]\{^e/_x\} \end{array}$$

- How it works:
  - The channel u is used to exchange a private name c
  - The meaning of u is that the receiver will be engaged in the rendez-vous with the sender
  - The sender confirms on c by sending the private channel d
  - Now the actual transmission can occur on the channel d

Polyadic π-calculus

- Monadic channels carry exactly one name: $\overline{u}\langle v \rangle$, $u(x)$
- Polyadic channels carry a vector of names:

$$
\begin{array}{lll}
P ::= & & \\
 & u(x_1, .., x_n).P & \text{input} \\
 & \overline{u}\langle v_1, .., v_n \rangle.P & \text{output}
\end{array}
$$

- We can communicate multiple values at the same time
- Reduction Rule:

$$\overline{a}\langle c_1, ..., c_n \rangle.P \mid a(x_1, ..., x_n).Q \longrightarrow P \mid Q\{^{\widetilde{c}}/_{\widetilde{x}}\}$$

Is there an encoding from polyadic to monadic channels?

Let's Try

- For every complex problem there is a simple solution ... that is wrong :)

$$\llbracket u(x_1,..,x_n).P \rrbracket = u(x_1).u(x_2)...u(x_n).\llbracket P \rrbracket$$
$$\llbracket \overline{u}\langle v_1,..,v_n\rangle.P \rrbracket = \overline{u}\langle v_1\rangle.\overline{u}\langle v_2\rangle...\overline{u}\langle v_n\rangle.\llbracket P \rrbracket$$

- Why the above encoding is ... wrong ?

- Hint: encode two processes in parallel sending on the same channel

$$\llbracket a(x_1,x_2).P_1 \mid a(x_3,x_4).P_2 \mid \overline{a}\langle c_1,c_2\rangle.P_3 \rrbracket$$

Encoding Polyadic π-calculus

$$[\![a(x_1, x_2).P_1]\!] \mid [\![a(x_3, x_4).P_2]\!] \mid [\![\overline{a}\langle c_1, c_2\rangle.P_3]\!] =$$
$$a(x_1).a(x_2).[\![P_1]\!] \mid a(x_3).a(x_4).[\![P_2]\!] \mid \overline{a}\langle c_1\rangle.\overline{a}\langle c_2\rangle.[\![P_3]\!] = R$$

- This reduction is fine:
  $R \longrightarrow [\![P]\!]\{^{c_1}/_{x_1}, {^{c_2}/_{x_2}}\} \mid a(x_3, x_4).[\![P_2]\!] \mid [\![P_3]\!]$

- But this is a mess:
  $R \longrightarrow a(x_2).[\![P_1]\!]\{^{c_1}/_{x_1}\} \mid a(x_4).[\![P_2]\!]\{^{c_2}/_{x_3}\} \mid [\![P_3]\!]$

Lets try again

Any Ideas?

Another approach

- Use new binding

– We need private channel for each tuple

$$
\begin{aligned}
[\![u(x_1,..,x_n).P]\!] &= u(z).z(x_1)...z(x_n).[\![P]\!] \\
[\![\overline{u}\langle v_1,..,v_n\rangle.P]\!] &= (\nu\, c)\overline{u}\langle c\rangle.\overline{c}\langle v_1\rangle...\overline{c}\langle v_n\rangle.[\![P]\!]
\end{aligned}
$$

– We still haven′t finished? We need a condition ...

Let's put it all together

	Monadic	Polyadic
Asynchronous	$\overline{u}\langle v\rangle$ $u(x).P$	$\bar{u}\langle v_1, \dots v_n\rangle$ $u(x_1, .. x_n).P_1$
Synchronous	$\overline{u}\langle v\rangle.P$ $u(x).P$	$\bar{u}\langle v_1, \dots v_n\rangle.P_1$ $u(x_1, .. x_n).P_1$

Adding more constructs ... Choice

In the asynchronous π-calculus there is no built-in choice operator $+$. Yet, we can represent *internal nondeterminism*.

$$P \oplus Q \stackrel{\text{df}}{=} (\nu\, a)(\overline{a} \,|\, a.P \,|\, a.Q) \quad \text{where } a \notin \mathit{fn}(P|Q)$$

There are two possible reductions:

- either $P \oplus Q \longrightarrow P \,|\, (\nu\, a)a.Q$
- or $P \oplus Q \longrightarrow (\nu\, a)a.P \,|\, Q$

Intuitively, since $(\nu\, a)a.Q$ and $(\nu\, a)a.P$ cannot reduce, the processes above are equivalent respectively to P and to Q.

Branching and Selection

- Structured external choice in the polyadic synchronous π-calculus:

$$P ::= ... \mid u \triangleright \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\} \mid u \triangleleft l.P$$

- We have labels (ranged over $l, l', ...$)

- The branching waits for the selector to select a label

- The reduction is:

$$a \triangleright \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\} \mid a \triangleleft l_k.P \longrightarrow P_k \mid P \quad (1 \leq k \leq n)$$

Is there an encoding into the polyadic synchronous π-calculus

Branching and selection

The encoding of branching and selection into the polyadic synchronous π-calculus is defined as follows.

- $[\![\mathbf{0}]\!] = \mathbf{0}$, $[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$, $[\![(\nu\, a)P]\!] = (\nu\, a)[\![P]\!]$, $[\![!P]\!] = ![\![P]\!]$,
- $[\![\overline{u}\langle\widetilde{v}\rangle]\!] = \overline{u}\langle\widetilde{v}\rangle$, $[\![u(\widetilde{x}).P]\!] = u(\widetilde{x}).[\![P]\!]$, and
- $$\begin{aligned} [\![u \triangleright \{l_1 : P_1 \ [\!]\ l_2 : P_2\}]\!] &= u(x).(\nu\, c_1, c_2)(\overline{x}\langle c_1, c_2\rangle \mid c_1.[\![P_1]\!] \mid c_2.[\![P_2]\!]) \\ [\![u \triangleleft l_1.P]\!] &= (\nu\, c)(\overline{u}\langle c\rangle \mid c(z_1, z_2).\overline{z_1}.[\![P]\!]) \\ [\![u \triangleleft l_2.P]\!] &= (\nu\, c)(\overline{u}\langle c\rangle \mid c(z_1, z_2).\overline{z_2}.[\![P]\!]) \end{aligned}$$
- We still haven't finished? We need a condition ...

Recursion

- We have
  - recursive definition $\mathbf{A}(\widetilde{x}) \stackrel{\text{df}}{=} Q$
  - a process P which uses this definition, by calling $\mathbf{A}\langle\widetilde{v}\rangle$
- How to encode that behaviour in the asynchronous π-calculus ?
- Theorem to the rescue:
  - Theorem: Any process involving recursive denitions is representable using replication, and conversely replication is redundant in the presence of recursion.
  - Tricky: we also need restriction

Recursion vs Replication

Using replication and restriction we can encode recursive definitions. Suppose we have the recursive definition $\mathbf{A}(\widetilde{x}) \stackrel{\text{df}}{=} Q$ and a process P which uses this definition, by calling $\mathbf{A}\langle\widetilde{v}\rangle$. We can encode this behaviour in the asynchronous π-calculus as follows:

1. choose a fresh channel name a not occurring in P or Q;
2. let P_a and Q_a be P and Q, where each recursive call of the form $\mathbf{A}\langle\widetilde{v}\rangle$ is replaced by an output process $\overline{a}\langle\widetilde{v}\rangle$;
3. replace P by $(\nu\, a)(P_a \,|\, !a(\widetilde{x}).Q_a)$

For example, consider the recursive definition and the reduction

$$\mathbf{BufferNext}(x) \stackrel{\text{df}}{=} x(y).(\overline{x}\langle y\rangle \,|\, \mathbf{BufferNext}\langle y\rangle)$$
$$\overline{b}\langle c\rangle \,|\, \mathbf{BufferNext}\langle b\rangle \longrightarrow \overline{b}\langle c\rangle \,|\, \mathbf{BufferNext}\langle c\rangle$$

with their non-recursive version

$$\mathbf{BufferNext_a} \stackrel{\text{df}}{=} x(y).(\overline{x}\langle y\rangle \,|\, \overline{a}\langle y\rangle)$$
$$(\nu\, a)(\overline{b}\langle c\rangle \,|\, \overline{a}\langle b\rangle \,|\, !a(x).\mathbf{BufferNext_a}) \stackrel{*}{\longrightarrow} (\nu\, a)(\overline{b}\langle c\rangle \,|\, \overline{a}\langle c\rangle \,|\, !a(x).\mathbf{BufferNext_a})$$