Message Passing

- CCS works well in modeling concurrency and synchronization among processes
- in concurrency we could also need to model message exchanges between processes
- A message is the object of a communication and it can be thought as the element of a certain data type (e.g., an integer, a string, an image, ...)
- How can we use a message?
  - Extensional view: the main aspect of the message is its value (used in logical-arithmetic operations)

    → **value passing CCS (vpCCS)**, where a message is simply an alpha-numeric datum
  - intensional view: the focus is on the functionalities that the message delivers to the receiver

    → **π-calculus:**
    - a new functionality is the possibility of being involved in a communication that was previously not possible because the communication channel was unknown
    - channels are at the same time the communication medium and the communication object
    - channels, that are the core elements of π-calculus, are called *names*

CCS / vpCCS / π-calculus

- Let us model a scenario where a client wants to buy a pizza
- In CCS, such a situation can be modeled by putting in parallel a client (process A) and the restaurant (process B), where:

$$A \triangleq \overline{askPizza}.\overline{pay}.pizza.EAT\_PIZZA \qquad B \triangleq askPizza.pay.\overline{pizza}$$

- This very basic interaction, in vpCCS can be enriched by several details, like, e.g., the kind of pizza and the relative amount of money:

$$\begin{aligned} A &\triangleq \overline{askPizza}\langle margherita\rangle.\overline{pay}\langle 4\ Euro\rangle.pizza.EAT\_PIZZA \\ B &\triangleq askPizza(x).pay(y).\textbf{if } y = price(x) \textbf{ then } \overline{pizza} \textbf{ else} \\ &\quad \textbf{if } y < price(x) \textbf{ then } \overline{askMoney} \textbf{ else } \overline{pizza}.\overline{output}\langle y - price(x)\rangle \end{aligned}$$

- Finally, in π-calculus we can also model the home delivery of the pizza:

$$\begin{aligned} A &\triangleq \overline{askPizza}\langle margherita, 4\ Euro, myHome\rangle.myHome(x,n).EAT(x) \\ B &\triangleq askPizza(x,y,z).\textbf{if } y < price(x) \textbf{ then } \overline{askMoney} \\ &\qquad \textbf{else } \overline{m}\langle pizza, y - price(x)\rangle \end{aligned}$$

Name creation

- A received name can be used for a communication, for forwarding it or for testing its value
- We'd like to create *new* names, i.e. names that are different from all the other names around
  - useful to protect the sent messages and model in this way secret channels
- New names can be seen as *local* channels (like the restricted names of CCS)
- Differently from restricted names, they can be communicated like normal names
  → Their scope changes during the computation (i.e., upon communications)

EXAMPLE:

- Alice that creates a restricted name for communicating with Bob:

  $A = (\nu c)a\langle c\rangle.c\langle\cdot\cdot\cdot\rangle$ $\qquad$ $B = a(x).x(\cdot\cdot\cdot)$
- After the creation and the transmission of the new channel c, Alice is ready to use this new channel and becomes $A' = c\langle\cdot\cdot\cdot\rangle$
- Bob is ready to receive information from c and it evolves to $B' = c(\cdots)$
  → after the communication, every occurrence of x in B has been replaced by c.
- Thus, the interaction between Alice and Bob leads to $(\nu c)(A' \mid B')$
- REMARK: the scope of c before the communication included only Alice, whereas after the communication also includes Bob

Syntax

- Let us assume a countable set of names N
- Notationally, a, b, c, ... are used to denote channel names, whereas x, y, z, ... denote input variables
- Syntax:

$$P \quad ::= \quad \mathbf{0} \mid a(x).P \mid \overline{a}\langle b\rangle.P \mid P_1|P_2 \mid (\nu a)P \mid [a=b]P \mid \, !P$$

- **0** is the inactive process, that performs no action;
- a(x) is the input prefix and a⟨b⟩ is the output prefix;
- | is the parallel operator (with an interleaving semantics);
- (va)P is the restriction, that makes a local to P;
- [a = b]P denotes name matching

  → it is a more compact way of writing if a = b then P ;
- !P is called replication and denotes an arbitrary number of parallel copies of P.

- REMARK: there is no non-deterministic choice (+); for our purposes, its presence is not fundamental but its absence reduces the expressive power of the calculus

- In processes a(x).P and (νa)P, names x and a are *bound*, with P the scope of such names. A name that is not bound is said *free*.
- fn(P) and bn(P) are the sets of the free and of the bound names of P, and n(P) all names of P

  (i.e., n(P) = fn(P) ∪ bn(P)).
- *Alpha-conversion*, denoted by $=_\alpha$, allows us to uniformly replace a bound name with a new (or fresh) one

  Example: (νd)a⟨d⟩.a(y).y(z).0 $=_\alpha$ (νc)a⟨c⟩.a(x).x(z).0.
- Usually, we shall omit trailing occurrences of **0**; for example, process a⟨b⟩.**0** will be usually written as a⟨b⟩.

Reduction Semantics vs LTS

- To define the operational semantics for the π-calculus, we have two possibilities:
  - via an LTS
  - via reductions
- An LTS describes all the possible actions a process can perform, both the visible and the invisible (viz., τ) ones → the LTS provides an "exhaustive" semantics (all process behaviors are provided, both the potential ones – i.e., the visible actions – and the actual ones – i.e., the τ's).
  - In order to develop an LTS, at least one rule for every construct is necessary. However, there are operators for which there are several rules (e.g., the parallel)
- Reductions only describe the actual computations of a term, i.e. those evolutions that are completely generated by the process under consideration
  - Disadvantage = loosing information about the process (i.e., its potential behaviors)
  - Advantage = we have less rules, that are also simpler

Structural Congruence

- The syntax is a way to write down a concurrent system.
  → the very same system can be written down in different ways
  (e.g., P|Q and Q|P)
- Thus, we first give a congruence that equates processes that define the same system
- Structural congruence, written ≡, is the smallest congruence including alpha- equivalence and satisfying the axioms:

(S-ID) $P \mid \mathbf{0} \equiv P$	(S-COM) $P \mid Q \equiv Q \mid P$
(S-ASS) $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	(S-EQ) $[a = a]P \equiv P$
(S-RCOM) $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$	(S-ABS) $(\nu a)\mathbf{0} \equiv \mathbf{0}$
(S-REP) $!P \equiv P \mid !P$	(S-EXT) $P \mid (\nu a)Q \equiv (\nu a)(P \mid Q) \quad \text{if } a \notin fn(P)$

(S-EXT)

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

- let us consider the following process:

  $Q = (\nu b)(b\langle c\rangle \mid b(x).Q1 \mid \ldots \mid b(x).Qn)$
- Name c can only be passed to Q1,...,Qn because name b is restricted.
- Let us now consider $P = b(x).P'$
- Without the side condition, we'd have

  $P \mid Q \equiv (\nu b)(b\langle c\rangle \mid b(x).Q1 \mid \ldots \mid b(x).Qn \mid b(x).P')$
- Hence, c could also be caught by P, against our initial aim

Reduction Rules

(R-Com)

$$a(x).P \mid \overline{a}\langle b\rangle.Q \longmapsto P[b/x] \mid Q$$

(R-Par)

$$\frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q}$$

(R-Res)

$$\frac{P \longmapsto P'}{(\nu a)P \longmapsto (\nu a)P'}$$

(R-Struct)

$$\frac{P \equiv Q \quad Q \longmapsto Q' \quad Q' \equiv P'}{P \longmapsto P'}$$

Example: $a(x).P|(\nu b)a\langle b\rangle.Q$, for $b \notin fn(P)$

$$\frac{a(x).P|(\nu b)\overline{a}\langle b\rangle.Q \;\equiv\; \dfrac{a(x).P|\overline{a}\langle b\rangle.Q \longmapsto P[b/x]|Q}{(\nu b)(a(x).P|\overline{a}\langle b\rangle.Q) \longmapsto (\nu b)(P[b/x]|Q)}}{a(x).P|(\nu b)\overline{a}\langle b\rangle.Q \longmapsto (\nu b)(P[b/x]|Q)}$$

Example: Secret channels

- Alice and Bob want to set up a secret channel, by using a forwarding server that shares with each of them a specific secret channel
  - c_{AS} is the secret channel between A and S
  - c_{BS} is the secret channel between B and S
  - c_{AB} is the secret channel that A wants to establish with B

$$
\begin{array}{lcl}
A & \triangleq & (\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle mess\rangle.A' \\
S & \triangleq & !c_{AS}(x).\overline{c_{BS}}\langle x\rangle \\
B & \triangleq & c_{BS}(z).z(w).B'
\end{array}
$$

- The overall system is $(\nu c_{AS}, c_{BS})(A|S|B)$ and it evolves as follows:

$$
\begin{array}{rl}
(\nu c_{AS}, c_{BS})(A|S|B) & \longmapsto (\nu c_{AS}, c_{BS}, c_{AB})(\overline{c_{AB}}\langle mess\rangle.A' \mid S \mid \overline{c_{BS}}\langle c_{AB}\rangle \mid B) \\
& \longmapsto (\nu c_{AS}, c_{BS}, c_{AB})(\overline{c_{AB}}\langle mess\rangle.A' \mid S \mid c_{AB}(w).B'[^{c_{AB}}/z]) \\
& \longmapsto (\nu c_{AS}, c_{BS}, c_{AB})(A' \mid S \mid B'[^{c_{AB}}/z, ^{mess}/w])
\end{array}
$$

Modeling Parametric Process Definitions

- We associate every process identifier A of arity k with a channel c_A, where we receive all the parameters of the invocation.

  → a process definition resembles an input, whereas
  a process invocation looks like an output.
- A process definition $A(x_1, \cdot\cdot\cdot, x_k) = P$ then becomes $!c_A(x_1,\cdots,x_k).P$

  → we allow the simultaneous reception of k names
- This process will be put in parallel with the remaining processes, translated by replacing every invocation $A\langle b_1, \cdots, b_k\rangle$ with the output $c_A\langle b_1,\cdots,b_k\rangle$

  → we allow the simultaneous sending of k names

- *Remark*: sending/receiving more names at once will be discussed in the next class (and shown to be implementable in the "one-name" basic framework)

Example

- Assume to have the proc. definition:

$$Snd(x_1, x_2, x_3) \triangleq \overline{x_1}\langle x_2\rangle.\overline{x_1}\langle x_3\rangle.Snd\langle x_1, x_2, x_3\rangle$$

- The proc. def. is translated to:

$$!\, c_{Snd}(x_1, x_2, x_3).\overline{x_1}\langle x_2\rangle.\overline{x_1}\langle x_3\rangle.\overline{c_{Snd}}\langle x_1, x_2, x_3\rangle$$

- Then, *Snd*⟨b, c, d⟩ is translated as follows:

$$\overline{c_{Snd}}\langle b, c, d\rangle$$

- Once we put the two in parallel, the resulting process reduces to:

$$!\, c_{Snd}(x_1, x_2, x_3).\overline{x_1}\langle x_2\rangle.\overline{x_1}\langle x_3\rangle.\overline{c_{Snd}}\langle x_1, x_2, x_3\rangle \;\mid\; \bar{b}\langle c\rangle.\bar{b}\langle d\rangle.\overline{c_{Snd}}\langle b, c, d\rangle$$

that behaves like *Snd*⟨b, c, d⟩

Making Maths in π-calculus

As usual,

$$n = succ^n(0) = 0 + \underbrace{1 + 1 + 1 ... + 1}_{n\ times}$$

In π-calculus, this can be modeled as follows:

$$\underline{n}_a \triangleq \overline{a}\langle u\rangle. \cdots .\overline{a}\langle u\rangle.\overline{a}\langle z\rangle$$

where z e u are reserved names that, respectively, represent 0 and 1.

We can now implement the successor function as follows::

$$Succ(a,b) \ \triangleq \ a(x).([x=z]\overline{b}\langle u\rangle.\overline{b}\langle z\rangle \mid \ [x=u]\overline{b}\langle u\rangle.Succ(a,b))$$

We can easily show that this implementation is correct, in particular that

$$(\nu a)(Succ(a,b) \mid \underline{n}_a) \ \approx \ \underline{n+1}_b$$

where $\approx$ is the weak bisimulation in the π-calculus.

The proof is by induction on n.

Base step $(n=0)$**:**

$$
\begin{aligned}
&(\nu a)(Succ(a,b) \mid \overline{a}\langle z\rangle) \\
&\qquad \approx\ (\nu a)([z=z]\overline{b}\langle u\rangle.\overline{b}\langle z\rangle \mid\ [z=u]\overline{b}\langle u\rangle.Succ(a,b)) \\
&\qquad \approx\ \overline{b}\langle u\rangle.\overline{b}\langle z\rangle\ \triangleq\ \underline{1}_b
\end{aligned}
$$

Inductive step:

$$
\begin{aligned}
&(\nu a)(Succ(a,b) \mid \overbrace{\overline{a}\langle u\rangle.\cdots.\overline{a}\langle u\rangle}^{n+1}.\overline{a}\langle z\rangle) \\
&\qquad \approx\ (\nu a)([u=u]\overline{b}\langle u\rangle.Succ(a,b) \mid \overbrace{\overline{a}\langle u\rangle.\cdots.\overline{a}\langle u\rangle}^{n}.\overline{a}\langle z\rangle) \\
&\qquad \equiv\ (\nu a)(\overline{b}\langle u\rangle.Succ(a,b) \mid \underbrace{\overline{a}\langle u\rangle.\cdots.\overline{a}\langle u\rangle}_{n}.\overline{a}\langle z\rangle) \\
&\qquad \approx\ \overline{b}\langle u\rangle.(\nu a)(Succ(a,b) \mid \underline{n}_a) \\
&\qquad \approx\ \overline{b}\langle u\rangle.\underline{n+1}_b\ \triangleq\ \overline{b}\langle u\rangle.\underbrace{\overline{b}\langle u\rangle.\cdots.\overline{b}\langle u\rangle}_{n+1}.\overline{b}\langle z\rangle\ \triangleq\ \underline{n+2}_b
\end{aligned}
$$