Models and Semantics of Computation

Jorge A. Pérez
Bernoulli Institute
University of Groningen

October 19, 2023

The π-calculus: A Calculus of Mobile Processes (I)

- informal introduction; contrast with CCS
- syntax and basic examples
- expressiveness (polyadic communication, recursion vs replication)

The π-calculus: A Calculus of Mobile Processes

Arguably, the paradigmatic calculus for concurrency

- Proposed by Milner, Parrow, and Walker in 1992.
  Developed significantly by Sangiorgi.

Interactive systems with dynamic connectivity (topology).
A dual role:

- A model of networked computation:
  Exchanged messages which contain links referring to communication channels themselves
- A basic model of computation:
  Interaction as the primitive notion of concurrent computing
  (Just as the λ-calculus for functional computing)

The π-calculus, in these lectures

- The theory of the π-calculus is richer than that of CCS.
  In some aspects, however, it is also more involved.
- We will overview this theory, contrasting it with CCS
- Hence, we present the π-calculus without going too much into technical details

Mobility as dynamic connectivity (1)

Towards the meaning of 'mobility':

- What kind of entity moves? In what space does it move?

Many possibilities—the two most relevant in this course are:

1. Processes move, in the virtual space of linked processes
2. Links move, in the virtual space of linked processes

Observe that

- A process' location is given by the links it has to other processes (think of your contacts in your mobile phone)
- Hence, the movement of a process can be represented by the movement of its links

Mobility as dynamic connectivity (1)

Towards the meaning of 'mobility':

- What kind of entity moves? In what space does it move?

Many possibilities—the two most relevant in this course are:

1. Processes move, in the virtual space of linked processes
2. Links move, in the virtual space of linked processes

Observe that

- A process' location is given by the links it has to other processes (think of your contacts in your mobile phone)
- Hence, the movement of a process can be represented by the movement of its links

Mobility as dynamic connectivity (2)

1. Processes move, in the virtual space of linked processes
2. Links move, in the virtual space of linked processes

The π-calculus commits to mobility in the sense of (2)...

- Economy, flexibility, and simplicity (at least wrt CCS)

...while models of higher-order concurrency stick to (1):

- Inspired in the λ-calculus
- It might be difficult/inconvenient to "normalize" all concurrency phenomena in the sense of (2)

We will argue that (1) and (2) need not be mutually exclusive

Mobility as dynamic connectivity (2)

1. Processes move, in the virtual space of linked processes
2. Links move, in the virtual space of linked processes

The π-calculus commits to mobility in the sense of (2)...

- Economy, flexibility, and simplicity (at least wrt CCS)

...while models of higher-order concurrency stick to (1):

- Inspired in the λ-calculus
- It might be difficult/inconvenient to "normalize" all concurrency phenomena in the sense of (2)

We will argue that (1) and (2) need not be mutually exclusive

Mobility as dynamic connectivity (2)

1. Processes move, in the virtual space of linked processes
2. Links move, in the virtual space of linked processes

The π-calculus commits to mobility in the sense of (2)...

- Economy, flexibility, and simplicity (at least wrt CCS)

...while models of higher-order concurrency stick to (1):

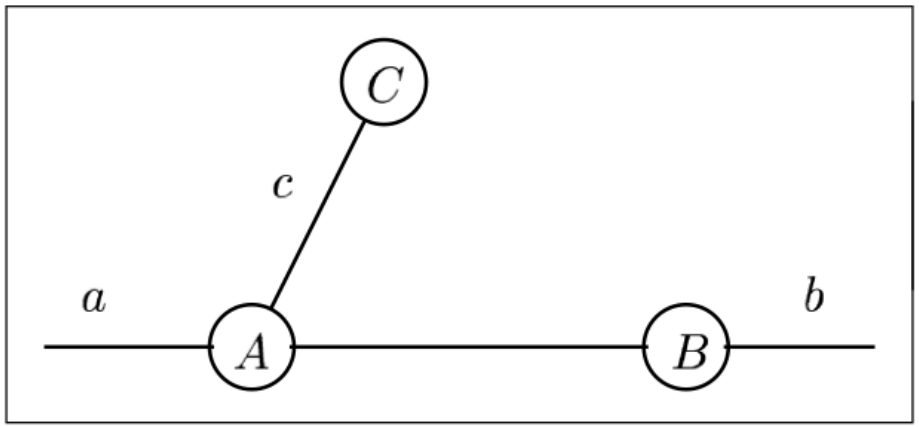
- Inspired in the λ-calculus
- It might be difficult/inconvenient to "normalize" all concurrency phenomena in the sense of (2)

We will argue that (1) and (2) need not be mutually exclusive

Dynamic connectivity in CCS is limited (1)

What's the main difference of the π-calculus wrt CCS?
The answer is **dynamic connectivity**.

Suppose a CCS process $S \stackrel{\text{def}}{=} (\nu x)(A \parallel C) \parallel B$.
Name a is free in A, while b is free in B. Graphically:

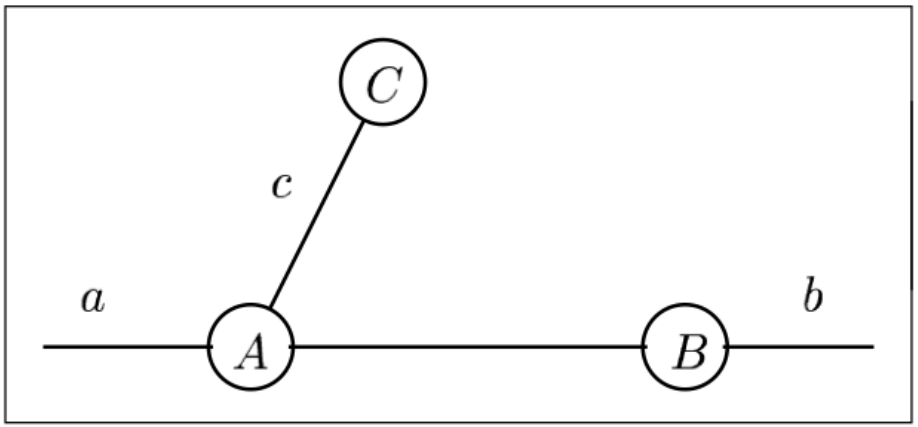


(1)

Dynamic connectivity in CCS is limited (1)

What's the main difference of the π-calculus wrt CCS?
The answer is **dynamic connectivity**.

Suppose a CCS process $S \stackrel{\text{def}}{=} (\nu x)(A \parallel C) \parallel B$.
Name a is free in A, while b is free in B. Graphically:

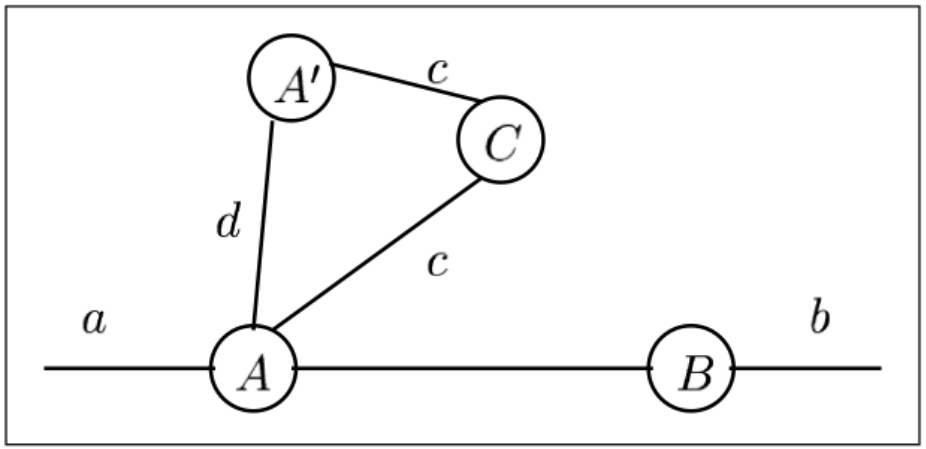


(1)

Dynamic connectivity in CCS is limited (2)

Suppose a CCS process $S \stackrel{\text{def}}{=} (\nu x)(A \parallel C) \parallel B$.
Name a is free in A, while b is free in B.

Suppose now that $A \stackrel{\text{def}}{=} a.(\nu d)(A \parallel A') + c.A''$. Graphically:



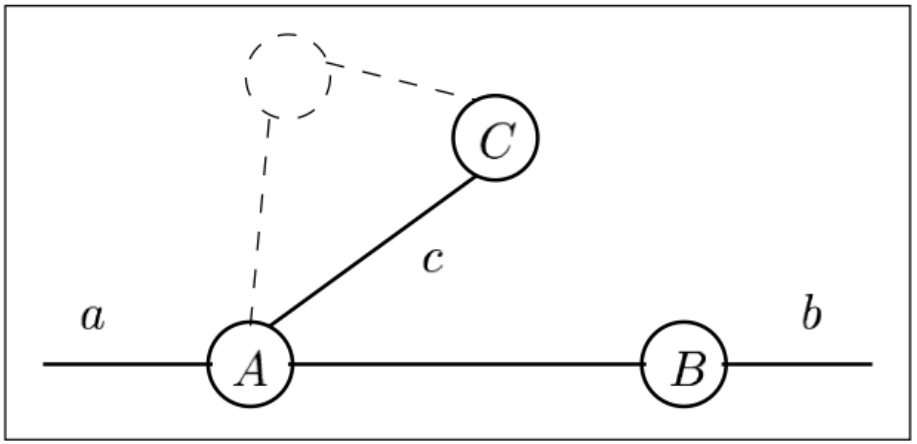
(2)

Dynamic connectivity in CCS is limited (3)

Suppose a CCS process $S \stackrel{\text{def}}{=} (\nu x)(A \parallel C) \parallel B$.
Name a is free in A, while b is free in B.

Suppose now that $A \stackrel{\text{def}}{=} a.(\nu d)(A \parallel A') + c.A''$.

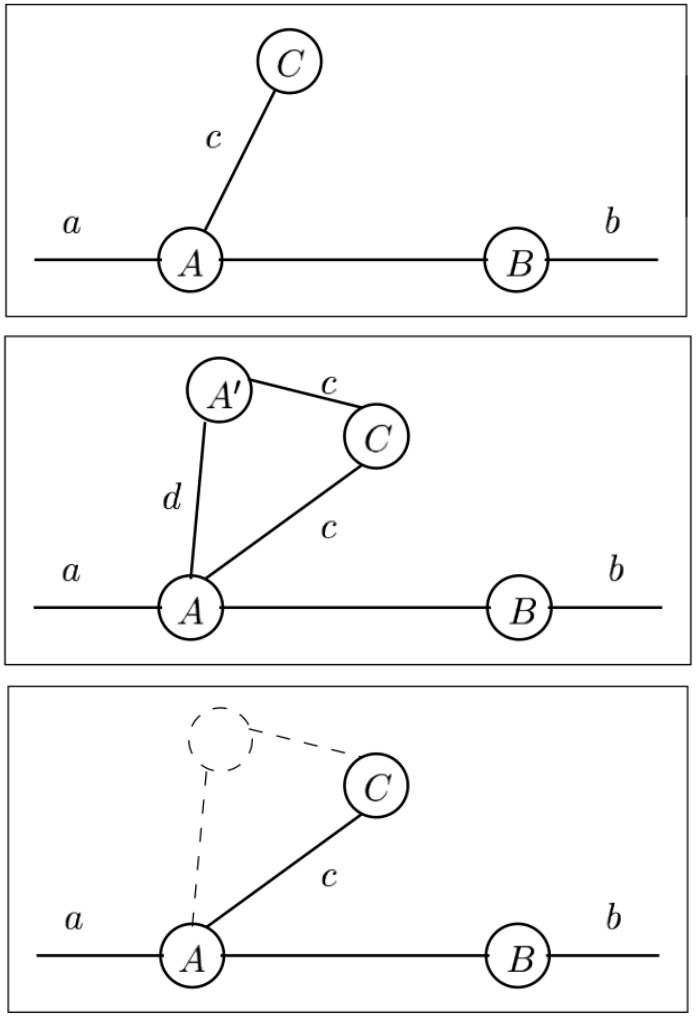
Finally, suppose that $A' = c.\mathbf{0}$. Process A' then dies. Graphically:



(3)

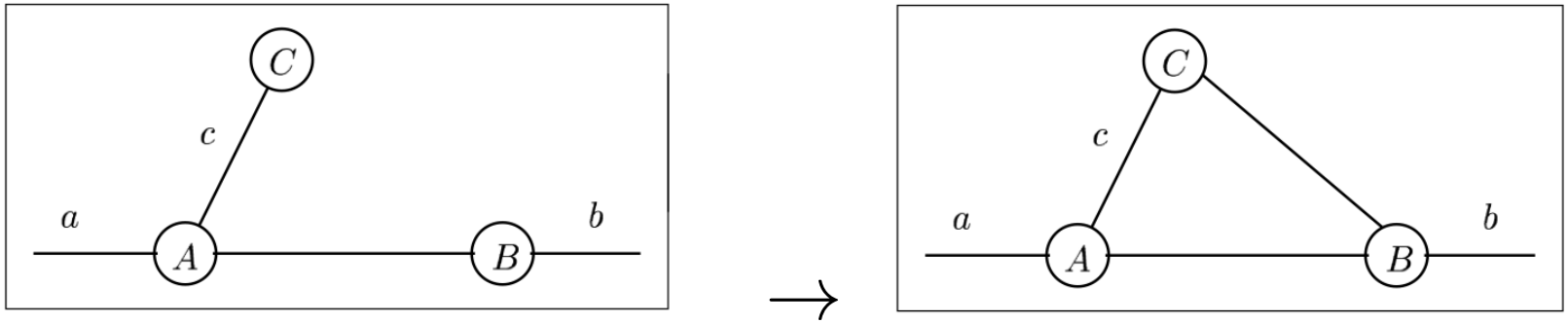
Dynamic connectivity in CCS is limited (4)

In CCS, links can proliferate and die:



Dynamic connectivity in CCS is limited (5)

However, new links between existing processes cannot be created.
The following transition is not possible in CCS:

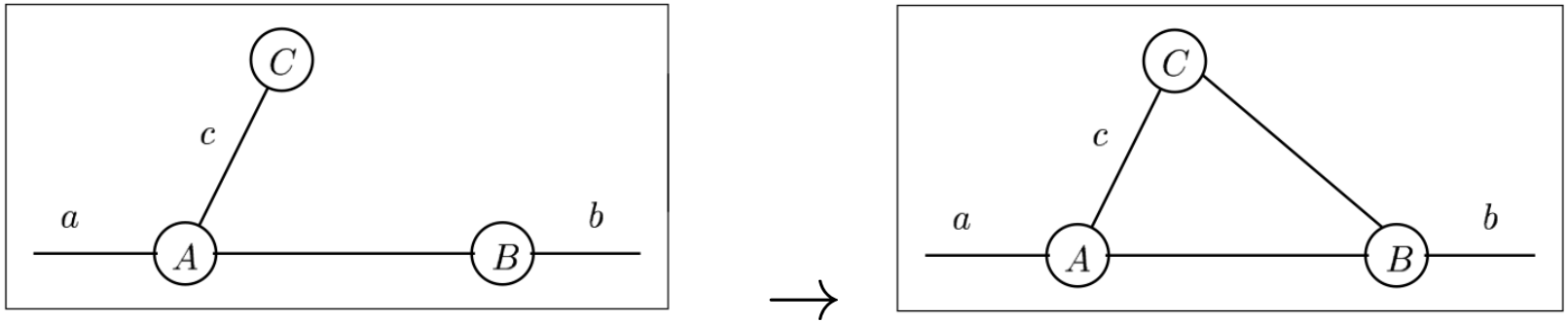


Dynamic connectivity refers to this kind of transitions.

The π-calculus goes beyond CCS by allowing dynamic communication topologies.

Dynamic connectivity in CCS is limited (5)

However, new links between existing processes cannot be created.
The following transition is not possible in CCS:



Dynamic connectivity refers to this kind of transitions.

The π-calculus goes beyond CCS by allowing dynamic communication topologies.

The π-calculus, more formally

We now formally introduce the π-calculus. Some highlights:

- The major novelty is communication of names
- Dynamic connectivity formalized as scope extrusion
- Two different semantics:
  - A reduction semantics, using a relation of structural congruence
  - A semantics based on labeled transitions (LTS)

  Roughly: a step in the reduction semantics, noted $P \longmapsto P'$, corresponds to a transition $P \xrightarrow{\tau} P'$ in the LTS (and vice versa).

The π-calculus, more formally

We use $x, y, z, \ldots$ to range over $\mathcal{N}$, an infinite set of names.

The action prefixes of the π-calculus generalize the actions of CCS:

$$\begin{array}{lcll} \alpha & ::= & \overline{x}\langle y \rangle & \text{send name } y \text{ along } x \\ & & x(y) & \text{receive a name along } x \\ & & \tau & \text{unobservable action} \end{array}$$

Brackets in $\overline{x}\langle y \rangle$ represent a tuple of values.

- Above, monadic communication: exactly one name is sent.
- In polyadic communication more than one value (a list of values) may be sent.

Process expressions of the π-calculus

P, Q	$::=$	$\mathbf{0}$	Inactive process
	$\mid$	$\alpha.P$	Prefix
	$\mid$	$P + P$	Sum
	$\mid$	$P \parallel Q$	Parallel composition
	$\mid$	$(\nu y)P$	Name creation/restriction
	$\mid$	$A\langle y_1, \ldots, y_n \rangle$	Identifier

We assume each identifier A is equipped with a recursive definition $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$, where $i \neq j$ implies $x_i \neq x_j$.

Free and bound names

- Restriction and input actions are name binders:
  In $(\nu y)\, P$ and $x(y).P$ name y is bound with scope P.
- In contrast, in $\overline{x}\langle y \rangle$ name y is free.

Structural Congruence: Intuitions

- The syntax of processes is too concrete: there are syntactically different terms that represent the "same behavior". Examples:

$$a(x).\overline{b}\langle x\rangle \quad \text{and} \quad a(y).\overline{b}\langle y\rangle$$
$$P \parallel Q \quad \text{and} \quad Q \parallel P$$

  [We often omit trailing $\mathbf{0}$s, and write $\overline{b}\langle y\rangle$ instead of $\overline{b}\langle y\rangle.\mathbf{0}$.]
- Structural congruence identifies processes which are "obviously the same" based on their syntactical structure
- In this sense, structural congruence will be stronger than any behavioral equivalence — that is, will equate less processes.

Structural Congruence

P and Q structurally congruent, written $P \equiv Q$, if we can transform one into the other by using the following equations:

1. α-conversion: change of bound names
2. Laws for parallel composition:

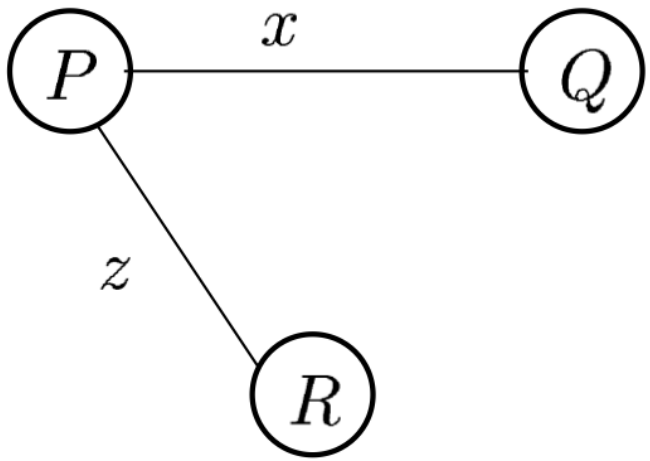
$$\begin{aligned} P \parallel \mathbf{0} &\equiv P \\ P \parallel Q &\equiv Q \parallel P \\ P \parallel (Q \parallel R) &\equiv (P \parallel Q) \parallel R \end{aligned}$$

3. Law for recursive definitions: $A\langle \tilde{y} \rangle \equiv P\{\tilde{y}/\tilde{x}\}$ if $A(\tilde{x}) \stackrel{\text{def}}{=} P$
4. Laws for restriction:

$$\begin{aligned} (\nu x)(P \parallel Q) &\equiv P \parallel (\nu x)Q \quad \text{if } x \notin \mathsf{fn}(P) \\ (\nu x)\mathbf{0} &\equiv \mathbf{0} \\ (\nu x)(P + Q) &\equiv P + (\nu x)Q \\ (\nu x)(\nu y)\, P &\equiv (\nu y)(\nu x)\, P \end{aligned}$$

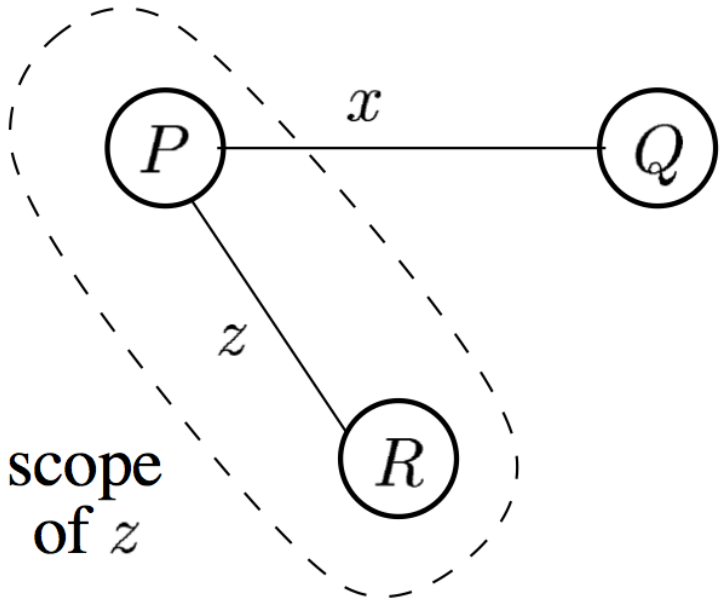
Scope Extrusion (1)

A process $P \parallel Q \parallel R$.
Name x is free in P and Q, while z is free in P and R:



Scope Extrusion (2)

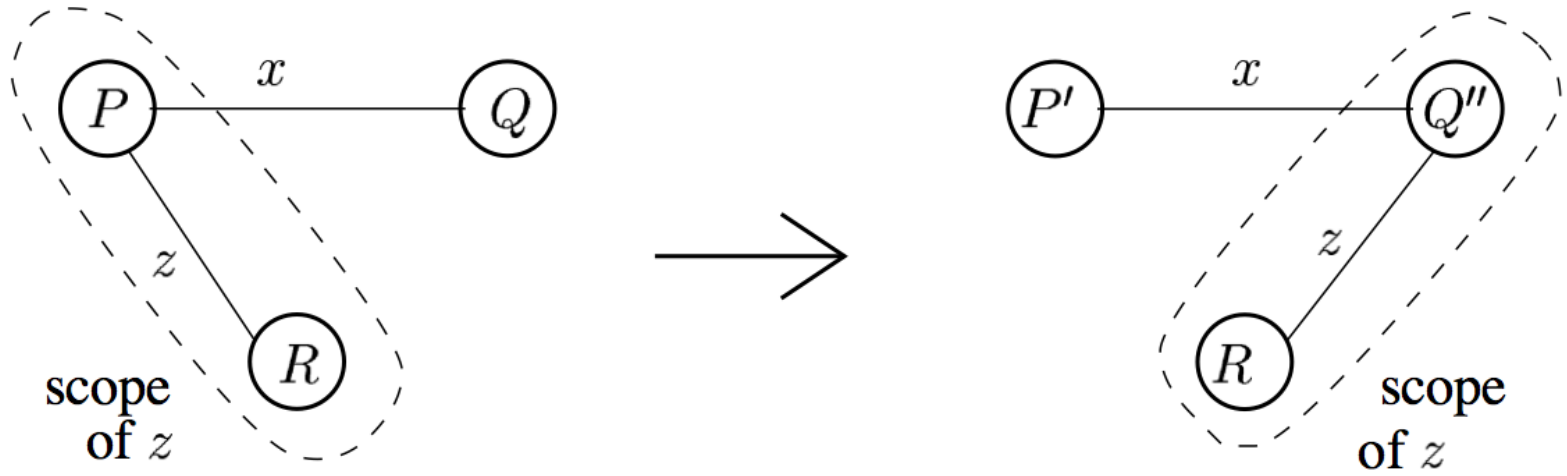
Suppose that z is restricted to P and R, while x is free in P and Q. That is, we have the process $(\nu z)(P \parallel R) \parallel Q$:



What happens if P wishes to send z to Q?

Scope Extrusion (3)

Suppose $P = \overline{x}\langle z \rangle.P'$, with $z \notin \mathrm{fn}(P')$.
Suppose also $Q = x(y).Q'$, with $z \notin \mathrm{fn}(Q')$.



where $Q'' = Q'\{z/y\}$. We have graphically described the reduction

$$(\nu z)(P \parallel R) \parallel Q \longmapsto P' \parallel (\nu z)(R \parallel Q'')$$

The above describes a movement of a way of accessing R (rather than a movement of R).

Initial Examples

We present some simple examples of scope extrusion.
For the moment, we use an informal account of reduction ($\longmapsto$), based on three postulates:

1. A law for inferring interactions:

$$a(x).P \parallel \overline{a}\langle b\rangle.Q \longmapsto P\{^{b}/_{x}\} \parallel Q$$

2. Restrictions respect silent transitions:

$$P \longmapsto Q \text{ implies } (\nu x)P \longmapsto (\nu x)Q$$

3. Structurally congruent processes should have the same behavior.

A Simple Example

We use str. congruence to infer an interaction for the process

$$a(x).\overline{c}\langle x\rangle \parallel (\nu b)\overline{a}\langle b\rangle$$

Since $b \notin \mathrm{fn}(a(x).\overline{c}\langle x\rangle)$, we have

$$a(x).\overline{c}\langle x\rangle \parallel (\nu b)\overline{a}\langle b\rangle \equiv (\nu b)(a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle)$$

We can infer that

$$(\nu b)(a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle) \longmapsto (\nu b)(\overline{c}\langle b\rangle \parallel \mathbf{0})$$

because $a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle \longmapsto \overline{c}\langle b\rangle \parallel \mathbf{0}$ is a valid interaction.

Removing $\mathbf{0}$, in general we have, for any $b \notin fn(P)$:

$$a(x).P \parallel (\nu b)\overline{a}\langle b\rangle.Q \longmapsto (\nu b)(P\{^{b}/_{x}\} \parallel Q)$$

and the scope of b has moved from the right to the left.

A Simple Example

We use str. congruence to infer an interaction for the process

$$a(x).\overline{c}\langle x\rangle \parallel (\nu b)\overline{a}\langle b\rangle$$

Since $b \notin \mathsf{fn}(a(x).\overline{c}\langle x\rangle)$, we have

$$a(x).\overline{c}\langle x\rangle \parallel (\nu b)\overline{a}\langle b\rangle \equiv (\nu b)(a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle)$$

We can infer that

$$(\nu b)(a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle) \longmapsto (\nu b)(\overline{c}\langle b\rangle \parallel \mathbf{0})$$

because $a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle \longmapsto \overline{c}\langle b\rangle \parallel \mathbf{0}$ is a valid interaction.

Removing $\mathbf{0}$, in general we have, for any $b \notin fn(P)$:

$$a(x).P \parallel (\nu b)\overline{a}\langle b\rangle.Q \longmapsto (\nu b)(P\{^{b}/_{x}\} \parallel Q)$$

and the scope of b has moved from the right to the left.

A Simple Example

We use str. congruence to infer an interaction for the process

$$a(x).\overline{c}\langle x\rangle \parallel (\nu b)\overline{a}\langle b\rangle$$

Since $b \notin \mathrm{fn}(a(x).\overline{c}\langle x\rangle)$, we have

$$a(x).\overline{c}\langle x\rangle \parallel (\nu b)\overline{a}\langle b\rangle \equiv (\nu b)(a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle)$$

We can infer that

$$(\nu b)(a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle) \longmapsto (\nu b)(\overline{c}\langle b\rangle \parallel \mathbf{0})$$

because $a(x).\overline{c}\langle x\rangle \parallel \overline{a}\langle b\rangle \longmapsto \overline{c}\langle b\rangle \parallel \mathbf{0}$ is a valid interaction.

Removing $\mathbf{0}$, in general we have, for any $b \notin fn(P)$:

$$a(x).P \parallel (\nu b)\overline{a}\langle b\rangle.Q \longmapsto (\nu b)(P\{^{b}/_{x}\} \parallel Q)$$

and the scope of b has moved from the right to the left.

Another Example

$$P = (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel x(u).\overline{u}\langle v\rangle \parallel \overline{x}\langle z\rangle)$$

Observe that $\mathsf{fn}(P) = \{x, v, y\}$ and $\mathsf{bn}(P) = \{z, w, u\}$.

There are two possibilities for reduction.

1. Interaction among the first and second components:

$$\begin{array}{rcl} P & \longmapsto & (\nu z)(\mathbf{0} \parallel \overline{u}\langle v\rangle\{y/u\} \parallel \overline{x}\langle z\rangle) \\ & = & (\nu z)(\mathbf{0} \parallel \overline{y}\langle v\rangle \parallel \overline{x}\langle z\rangle) = P_1 \end{array}$$

   Process $P\{y/u\}$ represents the process P in which the free occurrences of name u have been substituted with y.

2. Interaction among the second and third components:

$$\begin{array}{rcl} P & \longmapsto & (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{u}\langle v\rangle\{z/u\} \parallel \mathbf{0}) \\ & = & (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{z}\langle v\rangle \parallel \mathbf{0}) = P_2 \end{array}$$

While $P_1 \not\longmapsto$, we do have $P_2 \longmapsto (\nu z)(\overline{z}\langle y\rangle \parallel \mathbf{0} \parallel \mathbf{0}) \equiv (\nu z)\overline{z}\langle y\rangle$

Another Example

$$P = (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel x(u).\overline{u}\langle v\rangle \parallel \overline{x}\langle z\rangle)$$

Observe that $\mathsf{fn}(P) = \{x, v, y\}$ and $\mathsf{bn}(P) = \{z, w, u\}$.
There are two possibilities for reduction.

1. Interaction among the first and second components:

$$\begin{array}{rcl} P & \longmapsto & (\nu z)(\mathbf{0} \parallel \overline{u}\langle v\rangle\{^{y}/_{u}\} \parallel \overline{x}\langle z\rangle) \\ & = & (\nu z)(\mathbf{0} \parallel \overline{y}\langle v\rangle \parallel \overline{x}\langle z\rangle) = P_1 \end{array}$$

   Process $P\{^{y}/_{u}\}$ represents the process P in which the free occurrences of name u have been substituted with y.

2. Interaction among the second and third components:

$$\begin{array}{rcl} P & \longmapsto & (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{u}\langle v\rangle\{^{z}/_{u}\} \parallel \mathbf{0}) \\ & = & (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{z}\langle v\rangle \parallel \mathbf{0}) = P_2 \end{array}$$

While $P_1 \not\longmapsto$, we do have $P_2 \longmapsto (\nu z)(\overline{z}\langle y\rangle \parallel \mathbf{0} \parallel \mathbf{0}) \equiv (\nu z)\overline{z}\langle y\rangle$

Another Example

$$P = (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel x(u).\overline{u}\langle v\rangle \parallel \overline{x}\langle z\rangle)$$

Observe that $\mathrm{fn}(P) = \{x, v, y\}$ and $\mathrm{bn}(P) = \{z, w, u\}$.
There are two possibilities for reduction.

1. Interaction among the first and second components:

$$\begin{aligned} P &\longmapsto (\nu z)(\mathbf{0} \parallel \overline{u}\langle v\rangle\{y/u\} \parallel \overline{x}\langle z\rangle) \\ &= (\nu z)(\mathbf{0} \parallel \overline{y}\langle v\rangle \parallel \overline{x}\langle z\rangle) = P_1 \end{aligned}$$

   Process $P\{y/u\}$ represents the process P in which the free occurrences of name u have been substituted with y.

2. Interaction among the second and third components:

$$\begin{aligned} P &\longmapsto (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{u}\langle v\rangle\{z/u\} \parallel \mathbf{0}) \\ &= (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{z}\langle v\rangle \parallel \mathbf{0}) = P_2 \end{aligned}$$

While $P_1 \not\longmapsto$, we do have $P_2 \longmapsto (\nu z)(\overline{z}\langle y\rangle \parallel \mathbf{0} \parallel \mathbf{0}) \equiv (\nu z)\overline{z}\langle y\rangle$

Another Example

$$P = (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel x(u).\overline{u}\langle v\rangle \parallel \overline{x}\langle z\rangle)$$

Observe that $\mathrm{fn}(P) = \{x, v, y\}$ and $\mathrm{bn}(P) = \{z, w, u\}$.
There are two possibilities for reduction.

1. Interaction among the first and second components:

$$\begin{array}{rcl} P & \longmapsto & (\nu z)(\mathbf{0} \parallel \overline{u}\langle v\rangle\{^{y}/_{u}\} \parallel \overline{x}\langle z\rangle) \\ & = & (\nu z)(\mathbf{0} \parallel \overline{y}\langle v\rangle \parallel \overline{x}\langle z\rangle) = P_1 \end{array}$$

   Process $P\{^{y}/_{u}\}$ represents the process P in which the free occurrences of name u have been substituted with y.

2. Interaction among the second and third components:

$$\begin{array}{rcl} P & \longmapsto & (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{u}\langle v\rangle\{^{z}/_{u}\} \parallel \mathbf{0}) \\ & = & (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{z}\langle v\rangle \parallel \mathbf{0}) = P_2 \end{array}$$

While $P_1 \not\longmapsto$, we do have $P_2 \longmapsto (\nu z)(\overline{z}\langle y\rangle \parallel \mathbf{0} \parallel \mathbf{0}) \equiv (\nu z)\overline{z}\langle y\rangle$

Another Example

$$P = (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel x(u).\overline{u}\langle v\rangle \parallel \overline{x}\langle z\rangle)$$

Observe that $\mathrm{fn}(P) = \{x, v, y\}$ and $\mathrm{bn}(P) = \{z, w, u\}$.
There are two possibilities for reduction.

1. Interaction among the first and second components:

$$\begin{array}{rcl} P & \longmapsto & (\nu z)(\mathbf{0} \parallel \overline{u}\langle v\rangle\{^{y}/_{u}\} \parallel \overline{x}\langle z\rangle) \\ & = & (\nu z)(\mathbf{0} \parallel \overline{y}\langle v\rangle \parallel \overline{x}\langle z\rangle) = P_1 \end{array}$$

   Process $P\{^{y}/_{u}\}$ represents the process P in which the free occurrences of name u have been substituted with y.

2. Interaction among the second and third components:

$$\begin{array}{rcl} P & \longmapsto & (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{u}\langle v\rangle\{^{z}/_{u}\} \parallel \mathbf{0}) \\ & = & (\nu z)((\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle) \parallel \overline{z}\langle v\rangle \parallel \mathbf{0}) = P_2 \end{array}$$

While $P_1 \not\longmapsto$, we do have $P_2 \longmapsto (\nu z)(\overline{z}\langle y\rangle \parallel \mathbf{0} \parallel \mathbf{0}) \equiv (\nu z)\overline{z}\langle y\rangle$

Try it yourself

- Three agents: a printing server, a client, and a printer. The client wishes to print a document d.
- The client and the server share a public name b.
- The server and the printer share a name a.
- However, the client doesn't share names with the printer, so it cannot contact it.
- The document d cannot be transmitted along public names.

Try it yourself

The client, the server, and the printer in three different processes:

$$
\begin{aligned}
C = \;& (\nu r)\overline{b}\langle r\rangle.r(y). \\
& \quad (\nu s)\overline{y}\langle s\rangle.s(c).\overline{s}\langle d\rangle.\mathbf{0} \\
S = \;& b(x).\overline{x}\langle a\rangle.\mathbf{0} \\
P = \;& a(u).\overline{u}\langle c\rangle.u(w).\overline{print}\langle w\rangle.\mathbf{0} \\
N = \;& C \parallel S \parallel P
\end{aligned}
$$

- The client uses private names (r, s) to send requests: first to the server and then to the printer
- For simplicity, the server ends after receiving a client's request. By design of C, name a is exchanged on a private name (r).
- The printer sends a confirmation c after getting a request on a. All exchanges are along s, private to C and P.
- The entire network is the composition of the three processes

Try it yourself

The client, the server, and the printer in three different processes:

$$
\begin{aligned}
C = \;& (\nu r)\overline{b}\langle r\rangle.r(y). \\
& \quad (\nu s)\overline{y}\langle s\rangle.s(c).\overline{s}\langle d\rangle.\mathbf{0} \\
S = \;& b(x).\overline{x}\langle a\rangle.\mathbf{0} \\
P = \;& a(u).\overline{u}\langle c\rangle.u(w).\overline{print}\langle w\rangle.\mathbf{0} \\
N = \;& C \parallel S \parallel P
\end{aligned}
$$

- The client uses private names (r, s) to send requests: first to the server and then to the printer
- For simplicity, the server ends after receiving a client's request. By design of C, name a is exchanged on a private name (r).
- The printer sends a confirmation c after getting a request on a. All exchanges are along s, private to C and P.
- The entire network is the composition of the three processes

Basic Expressiveness of the π-calculus

We examine the basic expressiveness of the π-calculus. We show:

- How to represent polyadicity using monadic name passing
- The relationship between recursive definitions and replication

We appeal to encodings from one language to another to justify our claims.

Polyadic Communication

- For any $n \leq 0$, we would like to have prefixes

$$x(y_1, \cdots, y_n).P \quad \text{and} \quad \overline{x}\langle z_1, \cdots, z_n\rangle.Q \quad (\text{all } y_i \text{ are distinct})$$

- The naive encoding (ne) sends one name at the time:

$$\begin{array}{rcl} [\![x(y_1, \cdots y_n).P]\!]_{ne} & = & x(y_1).\cdots x(y_n).P \\ [\![\overline{x}\langle z_1, \cdots, z_n\rangle.Q]\!]_{ne} & = & \overline{x}\langle z_1\rangle.\cdots .\overline{x}\langle z_n\rangle.Q \end{array}$$

- Problem: this encoding is sensible to undesirable interferences. Take the process

$$S = x(y_1 y_2).P \parallel \overline{x}\langle z_1 z_2\rangle.Q \parallel \overline{x}\langle v_1 v_2\rangle.R$$

Using the encoding we get:

$$[\![S]\!]_{ne} = x(y_1).x(y_2).P \parallel \overline{x}\langle z_1\rangle.\overline{x}\langle z_2\rangle.Q \parallel \overline{x}\langle v_1\rangle.\overline{x}\langle v_2\rangle.R$$

$[\![S]\!]_{ne}$ features unintended behavior: 'mix-ups' in sent values.

Polyadic Communication

- For any $n \leq 0$, we would like to have prefixes

$$x(y_1, \cdots, y_n).P \quad \text{and} \quad \overline{x}\langle z_1, \cdots, z_n\rangle.Q \quad (\text{all } y_i \text{ are distinct})$$

- The naive encoding (ne) sends one name at the time:

$$\begin{aligned} [\![x(y_1, \cdots y_n).P]\!]_{ne} &= x(y_1).\cdots x(y_n).P \\ [\![\overline{x}\langle z_1, \cdots, z_n\rangle.Q]\!]_{ne} &= \overline{x}\langle z_1\rangle.\cdots .\overline{x}\langle z_n\rangle.Q \end{aligned}$$

- Problem: this encoding is sensible to undesirable interferences. Take the process

$$S = x(y_1 y_2).P \parallel \overline{x}\langle z_1 z_2\rangle.Q \parallel \overline{x}\langle v_1 v_2\rangle.R$$

Using the encoding we get:

$$[\![S]\!]_{ne} = x(y_1).x(y_2).P \parallel \overline{x}\langle z_1\rangle.\overline{x}\langle z_2\rangle.Q \parallel \overline{x}\langle v_1\rangle.\overline{x}\langle v_2\rangle.R$$

$[\![S]\!]_{ne}$ features unintended behavior: 'mix-ups' in sent values.

Polyadic Communication

- A correct encoding first establishes a fresh link in which values are sent, one by one
- More precisely, we define $[\![\cdot]\!]_{\mathsf{pc}} : \mathrm{poly}\pi \to \mathrm{mona}\pi$:

$$
\begin{array}{rcl}
[\![x(y_1 \cdots y_n).P]\!]_{\mathsf{pc}} & = & x(w).w(y_1).\cdots .w(y_n).[\![P]\!]_{\mathsf{pc}} \\
[\![\overline{x}\langle z_1 \cdots z_n\rangle.Q]\!]_{\mathsf{pc}} & = & (\nu w)(\overline{x}\langle w\rangle.\overline{w}\langle z_1\rangle.\cdots .\overline{x}\langle z_n\rangle.[\![Q]\!]_{\mathsf{pc}})
\end{array}
$$

  (the other operators are translated homomorphically)
- n-adic communication thus requires $n+1$ reductions.
- Question: would the above strategy work if the fresh name is created by the receiver?
- We say that polyadic communication is representable in the monadic π-calculus. The latter can be regarded as basic.

Polyadic Communication

- A correct encoding first establishes a fresh link in which values are sent, one by one
- More precisely, we define $[\![\cdot]\!]_{\mathsf{pc}} : \mathrm{poly}\pi \to \mathrm{mona}\pi$:

$$
\begin{array}{rcl}
[\![x(y_1 \cdots y_n).P]\!]_{\mathsf{pc}} & = & x(w).w(y_1).\cdots .w(y_n).[\![P]\!]_{\mathsf{pc}} \\
[\![\overline{x}\langle z_1 \cdots z_n\rangle.Q]\!]_{\mathsf{pc}} & = & (\nu w)(\overline{x}\langle w\rangle.\overline{w}\langle z_1\rangle.\cdots .\overline{x}\langle z_n\rangle.[\![Q]\!]_{\mathsf{pc}})
\end{array}
$$

  (the other operators are translated homomorphically)
- n-adic communication thus requires $n+1$ reductions.
- Question: would the above strategy work if the fresh name is created by the receiver?
- We say that polyadic communication is representable in the monadic π-calculus. The latter can be regarded as basic.

Recursive Definitions and Replication

We have seen the π-calculus with recursive definitions of the form

$$A(\tilde{x}) \stackrel{\text{def}}{=} Q_A$$

where

- A is a process identifier
- Q_A is a process expression which may contain calls of A and other parametric processes.

Another form of introducing infinite behavior is replication.
The replication of process P, written $!P$, is given by the definition

$$!P \stackrel{\text{def}}{=} P \parallel !P$$

That is, $!P$ represents an unbounded number of copies of P:

$$!P \equiv P \parallel !P \equiv P \parallel P \parallel !P \equiv P \parallel P \parallel P \parallel !P \cdots$$

Recursive definitions and replication can be mutually represented!

Recursive Definitions and Replication

We have seen the π-calculus with recursive definitions of the form

$$A(\tilde{x}) \stackrel{\text{def}}{=} Q_A$$

where

- A is a process identifier
- Q_A is a process expression which may contain calls of A and other parametric processes.

Another form of introducing infinite behavior is replication.
The replication of process P, written $!P$, is given by the definition

$$!P \stackrel{\text{def}}{=} P \parallel !P$$

That is, $!P$ represents an unbounded number of copies of P:

$$!P \equiv P \parallel !P \equiv P \parallel P \parallel !P \equiv P \parallel P \parallel P \parallel !P \cdots$$

Recursive definitions and replication can be mutually represented!

Recursive Definitions and Replication

We have seen the π-calculus with recursive definitions of the form

$$A(\tilde{x}) \stackrel{\text{def}}{=} Q_A$$

where

- A is a process identifier
- Q_A is a process expression which may contain calls of A and other parametric processes.

Another form of introducing infinite behavior is replication.
The replication of process P, written $!P$, is given by the definition

$$!P \stackrel{\text{def}}{=} P \parallel !P$$

That is, $!P$ represents an unbounded number of copies of P:

$$!P \equiv P \parallel !P \equiv P \parallel P \parallel !P \equiv P \parallel P \parallel P \parallel !P \cdots$$

Recursive definitions and replication can be mutually represented!

Encoding Replication into Recursive Definitions

- From the previous description, it is clear that replication is a particular case of recursive definition.
- So the encoding formalizes a simple idea: each $!P$ is translated into a process A_P, recursively defined as

$$A_P(\tilde{x}) \stackrel{\text{def}}{=} P \parallel A_P\langle\tilde{x}\rangle$$

which can provide an unbounded number of copies of P.

- More formally: let $\pi_!$ and π_{rd} denote the set of π-calculus expressions with replication and recursive definitions, resp. Let $[\![\cdot]\!]_{\text{rep}} : \pi_! \to \pi_{\text{rd}}$ be the encoding defined as

$$[\![!P]\!]_{\text{rep}} \quad = \quad A\langle\tilde{x}\rangle \quad \text{where } A(\tilde{x}) \stackrel{\text{def}}{=} P \parallel A\langle\tilde{x}\rangle \text{ with } \tilde{x} \subseteq \text{fn}(P)$$

(the other operators are translated homomorphically)

Encoding Replication into Recursive Definitions

- From the previous description, it is clear that replication is a particular case of recursive definition.
- So the encoding formalizes a simple idea: each $!P$ is translated into a process A_P, recursively defined as

$$A_P(\tilde{x}) \stackrel{\text{def}}{=} P \parallel A_P\langle\tilde{x}\rangle$$

  which can provide an unbounded number of copies of P.
- More formally: let $\pi_!$ and π_{rd} denote the set of π-calculus expressions with replication and recursive definitions, resp. Let $[\![\cdot]\!]_{\mathsf{rep}} : \pi_! \to \pi_{\mathsf{rd}}$ be the encoding defined as

$$[\![!P]\!]_{\mathsf{rep}} \quad = \quad A\langle\tilde{x}\rangle \quad \text{where } A(\tilde{x}) \stackrel{\text{def}}{=} P \parallel A\langle\tilde{x}\rangle \text{ with } \tilde{x} \subseteq \mathsf{fn}(P)$$

  (the other operators are translated homomorphically)

Encoding Recursive Definitions into Replication

The encoding in the other direction is a bit more interesting.

Intuitively, for each definition $A(\tilde{x}) \stackrel{\text{def}}{=} Q_A$ we use

- message passing to model the passing of parameters
- replication to model the multiple activations
- restriction to model the scope of the definition

Encoding Recursive Definitions into Replication

In two steps, we set up a "server" for each recursive definition.

First, we define the encoding $[\![\cdot]\!]_{\mathsf{rd}_0} : \pi_{\mathsf{rd}} \to \pi_!$:

$$\begin{array}{rcl} [\![A_i(\tilde{x}_i) \stackrel{\text{def}}{=} Q_i]\!]_{\mathsf{rd}_0} & = & !a_i(\tilde{x}_i).[\![Q_i]\!]_{\mathsf{rd}_0} \\ [\![A\langle\tilde{z}\rangle]\!]_{\mathsf{rd}_0} & = & \overline{a_i}\langle\tilde{z}\rangle \end{array}$$

(the other constructs are translated homomorphically)

Then, we treat processes which contain identifiers.

Let P be a process with $A_1(\tilde{x_1}) \stackrel{\text{def}}{=} Q_1, \ldots, A_k(\tilde{x_k}) \stackrel{\text{def}}{=} Q_k$ as the sets of definitions of its process identifiers. We have:

$$[\![P]\!]_{\mathsf{rd}} \quad = \quad (\nu a_1, \ldots, a_k)([\![P]\!]_{\mathsf{rd}_0} \parallel \prod_{i \in \{1,\ldots,k\}} [\![A_i(\tilde{x}_i) \stackrel{\text{def}}{=} Q_i]\!]_{\mathsf{rd}_0})$$

where $a_1, \ldots, a_k \notin \mathsf{fn}(P)$.

Encoding Recursive Definitions into Replication

In two steps, we set up a "server" for each recursive definition.
First, we define the encoding $[\![\cdot]\!]_{\mathsf{rd}_0} : \pi_{\mathsf{rd}} \to \pi_!$:

$$\begin{array}{rcl} [\![A_i(\tilde{x}_i) \stackrel{\text{def}}{=} Q_i]\!]_{\mathsf{rd}_0} & = & !a_i(\tilde{x}_i).[\![Q_i]\!]_{\mathsf{rd}_0} \\ [\![A\langle\tilde{z}\rangle]\!]_{\mathsf{rd}_0} & = & \overline{a_i}\langle\tilde{z}\rangle \end{array}$$

(the other constructs are translated homomorphically)

Then, we treat processes which contain identifiers.
Let P be a process with $A_1(\tilde{x}_1) \stackrel{\text{def}}{=} Q_1, \ldots, A_k(\tilde{x}_k) \stackrel{\text{def}}{=} Q_k$ as the sets of definitions of its process identifiers. We have:

$$[\![P]\!]_{\mathsf{rd}} \quad = \quad (\nu a_1, \ldots, a_k)([\![P]\!]_{\mathsf{rd}_0} \parallel \prod_{i \in \{1,\ldots,k\}} [\![A_i(\tilde{x}_i) \stackrel{\text{def}}{=} Q_i]\!]_{\mathsf{rd}_0})$$

where $a_1, \ldots, a_k \notin \mathsf{fn}(P)$.

Encoding Recursive Definitions into Replication

In two steps, we set up a "server" for each recursive definition.
First, we define the encoding $[\![\cdot]\!]_{\mathsf{rd}_0} : \pi_{\mathsf{rd}} \to \pi_!$:

$$\begin{array}{rcl} [\![A_i(\tilde{x}_i) \stackrel{\text{def}}{=} Q_i]\!]_{\mathsf{rd}_0} & = & !a_i(\tilde{x}_i).[\![Q_i]\!]_{\mathsf{rd}_0} \\ [\![A\langle\tilde{z}\rangle]\!]_{\mathsf{rd}_0} & = & \overline{a_i}\langle\tilde{z}\rangle \end{array}$$

(the other constructs are translated homomorphically)

Then, we treat processes which contain identifiers.
Let P be a process with $A_1(\tilde{x_1}) \stackrel{\text{def}}{=} Q_1, \ldots, A_k(\tilde{x_k}) \stackrel{\text{def}}{=} Q_k$ as the sets of definitions of its process identifiers. We have:

$$[\![P]\!]_{\mathsf{rd}} \quad = \quad (\nu a_1, \ldots, a_k)([\![P]\!]_{\mathsf{rd}_0} \parallel \prod_{i \in \{1,\ldots,k\}} [\![A_i(\tilde{x}_i) \stackrel{\text{def}}{=} Q_i]\!]_{\mathsf{rd}_0})$$

where $a_1, \ldots, a_k \notin \mathsf{fn}(P)$.

Recursive Definitions and Replication

- The two encodings demonstrate that infinite behavior in the π-calculus is a natural concept
- We can choose the form of infinite behavior that fits better in our models
- Interestingly, a similar result does not hold for CCS