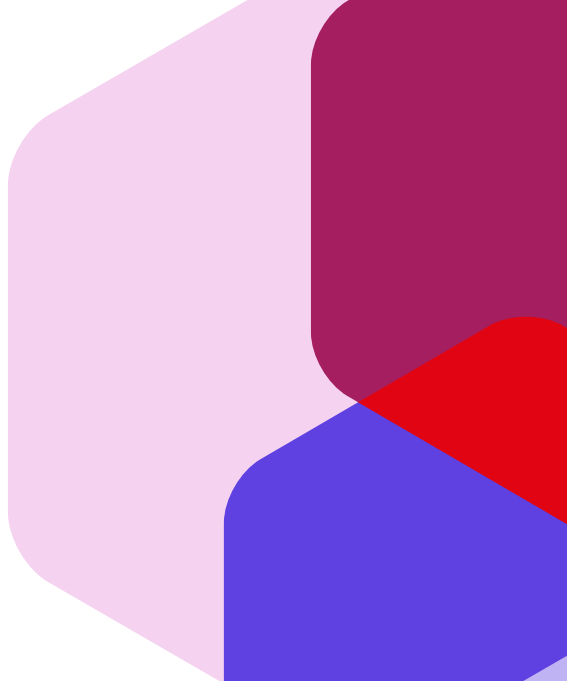


Principles, Algorithms and Models of Concurrency and Synchronization

The Pi-Calculus



We start with a general introduction to semantics:

- Understand the notions: syntax, semantics, contextual equivalence
- An exemplary operational semantics for a **sequential** programming language:

The λ -calculus

Afterwards: semantics of concurrent languages

- We consider **program calculi** as models for programming languages
- They can be understood as **core languages** of such programming languages
- Models of the programming languages, but easier to handle mathematically
- A calculus consists of **syntax** and **semantics**

Syntax

- Specifies which programs (expressions) may be formed
- Which **constructs** does the calculus provide?

Semantics

- Specifies the **meaning** of the programs
- The field of **formal semantics** knows different approaches:
 - Axiomatic semantics
 - Denotational semantics
 - Operational semantics

Axiomatic semantics

- Description of properties of programs by means of **logical axioms** and **inference rules**
- **Derivation** of new properties using the inference rules
- Prominent example: Hoare logic, e.g.
Hoare triple $\{P\}S\{Q\}$: precondition P , program S , postcondition Q
Inference rule, e.g.:

$$\text{Sequence: } \frac{\{P\}S\{Q\}, \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

- In general captures only some properties of programs, not all

Denotational semantics

- Mapping of programs into mathematical spaces by a semantic function
- Often uses partially ordered sets (domains)
- In the non-deterministic case: power domains (i.e. sets of sets)
- E.g. $\llbracket \cdot \rrbracket$ as semantic function:

$$\llbracket \text{if } a \text{ then } b \text{ else } c \rrbracket = \begin{cases} \llbracket b \rrbracket, & \text{if } \llbracket a \rrbracket = \text{True} \\ \llbracket c \rrbracket, & \text{if } \llbracket a \rrbracket = \text{False} \\ \perp, & \text{otherwise} \end{cases}$$

- Generally regarded as mathematically elegant
- Difficult for many constructs
- Very difficult for non-determinism

Operational semantics

- defines exactly the **evaluation/execution** of programs
- defines, so to speak, an interpreter
- Various formalisms:
 - State transition systems
 - Abstract machines
 - Rewriting systems
- Distinction into **small-step** and **big-step** semantics
- We use operational semantics

- As an introductory example we consider the **lambda calculus**
- Model for **sequential** programming languages
- In particular for **functional** programming languages like Haskell
- Introduced by Alonzo Church in the 1930s
- The lambda calculus is **Turing**-complete.

Syntax of the Lambda Calculus

Expr	::=	V	variable (infinite set)
		$ \lambda V. \mathbf{Expr}$	abstraction
		$ (\mathbf{Expr} \mathbf{Expr})$	application

- $\lambda x.s$: x is **bound** in s
- Abstractions are **anonymous functions**
 $id(x) = x$ in lambda notation $\lambda x.x$
- $(s \ t)$ allows the application of functions to arguments
- s, t may be arbitrary expressions
- hence **higher-order lambda calculus**
- E.g.: $id(id)$ can in lambda notation be: $(\lambda x.x) (\lambda x.x)$

- $FV(t)$: free variables of t
- $BV(t)$: bound variables of t

$$\begin{aligned}FV(x) &= x \\FV(\lambda x.s) &= FV(s) \setminus \{x\} \\FV(s\ t) &= FV(s) \cup FV(t)\end{aligned}$$

$$\begin{aligned}BV(x) &= \emptyset \\BV(\lambda x.s) &= BV(s) \cup \{x\} \\BV(s\ t) &= BV(s) \cup BV(t)\end{aligned}$$

- $FV(t) = \emptyset \implies t$ **closed**, t **program**, otherwise t **open**

- $s[t/x]$ = expression that results after **replacing** all **free occurrences** of x in s by t
- Avoiding name clashes in doing so: $x \notin BV(s)$

$$\begin{aligned}x[t/x] &= t \\y[t/x] &= y, \text{ if } x \neq y \\(\lambda y.s)[t/x] &= \lambda y.(s[t/x]) \\(s_1 \ s_2)[t/x] &= (s_1[t/x] \ s_2[t/x])\end{aligned}$$

$$\text{E.g. } (\lambda x.z \ x)[(\lambda y.y)/z] = (\lambda x.((\lambda y.y) \ x))$$

- **Context** = expression that has a **hole** $[\cdot]$ at one position instead of a subexpression

As a grammar:

$$C ::= [\cdot] \mid \lambda V.C \mid (C \text{ Expr}) \mid (\text{Expr } C)$$

- Expressions can be **plugged** into contexts:
context C , expression s :
 $C[s]$ yields the expression in which the hole in C is replaced by s
- Example: $C = ([\cdot] (\lambda x.x))$, then: $C[\lambda y.y] = ((\lambda y.y) (\lambda x.x))$.
- Plugging in may/can capture free variables,
e.g. $C = (\lambda x.[\cdot])$, then $C[\lambda y.x] = (\lambda x.\lambda y.x)$.

Alpha renaming step

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \text{ if } y \notin BV(\lambda x.s) \cup FV(\lambda x.s)$$

Alpha-equivalence

$=_{\alpha}$ is the **reflexive-transitive closure** of $\xrightarrow{\alpha}$

- We consider α -equivalent expressions to be **equal**.
- e.g. $\lambda x.x =_{\alpha} \lambda y.y$
- Distinct Variable Convention: All bound variables are distinct, and bound variables are distinct from free ones.
- α -renamings make it possible that the DVC can always be satisfied.

Beta reduction

$$(\beta) \quad (\lambda x.s) t \rightarrow s[t/x]$$

- If $s_0 = (\lambda x.s) t \xrightarrow{\beta} s[t/x] = t_0$, then one says: s_0 reduces immediately to t_0
- To fix the operational semantics, one still has to define where the β -reduction is applied
- Consider $((\lambda x.xx)((\lambda y.y)(\lambda z.z)))$, then:

$$\begin{aligned} & \underline{((\lambda x.xx)((\lambda y.y)(\lambda z.z)))} \rightarrow ((\lambda y.y)(\lambda z.z)) ((\lambda y.y)(\lambda z.z)) \text{ or} \\ & ((\lambda x.xx) \underline{((\lambda y.y)(\lambda z.z))}) \rightarrow ((\lambda x.xx)(\lambda z.z)). \end{aligned}$$

Definition

Call-by-name **reduction contexts** R :

$$R ::= [\cdot] \mid (R \text{ Expr})$$

If $s_0 \xrightarrow{\beta} t_0$, then

$$R[s_0] \xrightarrow{\text{name}} R[t_0]$$

is a **call-by-name reduction step**

Example

$$\begin{aligned} ((\lambda x. (x \ x)) \ (\lambda y. y)) \ ((\lambda w. w) \ (\lambda z. (z \ z))) &\xrightarrow{\beta} (x \ x)[(\lambda y. y)/x] \ ((\lambda w. w) \ (\lambda z. (z \ z))) \\ &= ((\lambda y. y) \ (\lambda y. y)) \ ((\lambda w. w) \ (\lambda z. (z \ z))) \end{aligned}$$

here $R = ([\cdot] \ ((\lambda w. w) \ \lambda z. (z \ z)))$

Call-by-Name Reduction (2)

- Call-by-name reduction is **deterministic**: For every expression s there is at most one expression t such that $s \xrightarrow{\text{name}} t$.
- There are also expressions for which no reduction is possible:
 - Reduction hits a free variable: e.g. $(x (\lambda y.y))$
 - Expression is a **value**: value = abstraction
- $\xrightarrow{\text{name},+} =$ transitive closure of $\xrightarrow{\text{name}}$
- $\xrightarrow{\text{name},*} =$ reflexive-transitive closure of $\xrightarrow{\text{name}}$

Definition

An expression s (call-by-name) **converges** ($s \downarrow_{\text{name}}$) iff $\exists$ abstraction $v : s \xrightarrow{\text{name},*} v$.

Otherwise s **diverges**, notation $s \uparrow_{\text{name}}$

- Implementations use the call-by-need variant:
avoiding double evaluations
- Call-by-name (and also call-by-need) evaluation are optimal with respect to convergence:

Statement

Let s be a lambda expression and let s be reducible to an abstraction v using arbitrary β -reductions (at arbitrary positions). Then $s \downarrow_{name}$ holds.

So far:

- Call-by-name lambda calculus: expressions, $\xrightarrow{\text{name}}$, $\downarrow_{\text{name}}$

I.e. syntax + operational semantics.

Still missing:

- Notion: when are two expressions equal
- I.e. in particular: when may a compiler replace one expression by another?

- Leibniz's principle: two things are equal if they have the same properties, with respect to all properties.
- For calculi: two expressions s, t are equal if they cannot be distinguished, no matter in which context they are used.
- More formally: s and t are equal if for all C : $C[s]$ and $C[t]$ behave the same.
- Behavior still has to be defined. For deterministic languages, observing termination suffices

- Leibniz's principle: two things are equal if they have the same properties, with respect to all properties.
- For calculi: two expressions s, t are equal if they cannot be distinguished, no matter in which context they are used.
- More formally: s and t are equal if for all C : $C[s]$ and $C[t]$ behave the same.
- Behavior still has to be defined. For deterministic languages, observing termination suffices

- Leibniz's principle: two things are equal if they have the same properties, with respect to all properties.
- For calculi: two expressions s, t are equal if they cannot be distinguished, no matter in which context they are used.
- More formally: s and t are equal if for all C : $C[s]$ and $C[t]$ behave the same.
- Behavior still has to be defined. For deterministic languages, observing termination suffices

- Leibniz's principle: two things are equal if they have the same properties, with respect to all properties.
- For calculi: two expressions s, t are equal if they cannot be distinguished, no matter in which context they are used.
- More formally: s and t are equal if for all C : $C[s]$ and $C[t]$ behave the same.
- Behavior still has to be defined. For deterministic languages, observing termination suffices

Contextual approximation and equivalence

- $s \leq_{c, name} t$ iff $\forall C : C[s] \downarrow_{name} \implies C[t] \downarrow_{name}$
- $s \sim_{c, name} t$ iff $s \leq_{c, name} t$ and $t \leq_{c, name} s$

Equivalence (4)

- $\sim_{c,name}$ is a **congruence**
- **Congruence** = equivalence relation that is compatible with contexts:
$$s \sim t \implies C[s] \sim C[t].$$
- Proving equivalence is generally hard, disproving generally easy.

Equivalence (4)

- $\sim_{c,name}$ is a **congruence**
- **Congruence** = equivalence relation that is compatible with contexts:
$$s \sim t \implies C[s] \sim C[t].$$
- Proving equivalence is generally hard, disproving generally easy.

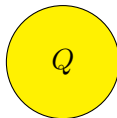
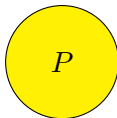
Example equivalence: $(\beta) \subseteq \sim_{c,name}$

- The π -calculus is a model of concurrent and distributed programming languages
- Data and operations are expressed by processes
- Control flow is expressed by process communication.
- In other words, it is a message-passing model: communication only via messages
- Special feature: processes are mobile

The π -Calculus: History

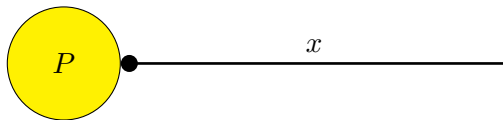
- Developed by Robin Milner, Joachim Parrow, and David Walker in the 1990s
- Applications also outside concurrent programming even outside computer science, e.g.
 - spi-calculus, applied pi-calculus:
Variants of π -calculus for describing and verifying cryptographic protocols
 - Microsoft's XLANG: Description language for business processes
 - Biochemistry: Stochastic π -calculus for the formal representation of biochemical processes

- Synchronous π -calculus
- Asynchronous π -calculus
- With or without sums: Non-deterministic selection
- With recursion / with replication
- Monadic / polyadic
- etc.



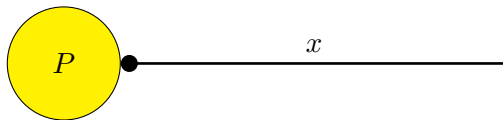
$$P \mid Q$$

“Processes P and Q run concurrently”



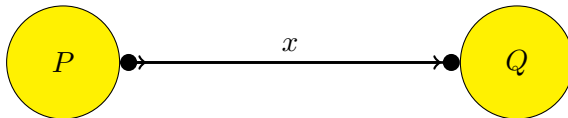
$$x.P \quad \text{or} \quad \bar{x}.P$$

" P is connected to channel named x "



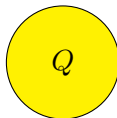
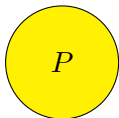
$$\underbrace{x.P}_{\text{receive}} \quad \text{or} \quad \underbrace{\bar{x}.P}_{\text{send}}$$

" P is connected to channel named x "



$$\bar{x}.P \mid x.Q$$

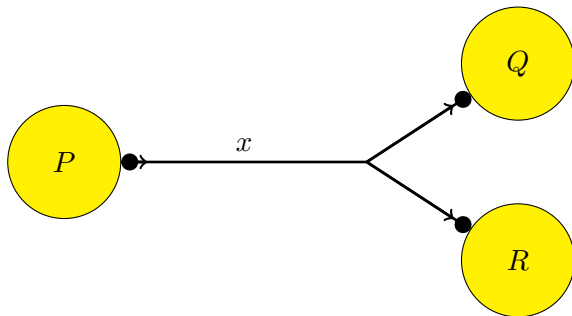
“ P (sender) and Q (receiver)
can communicate”



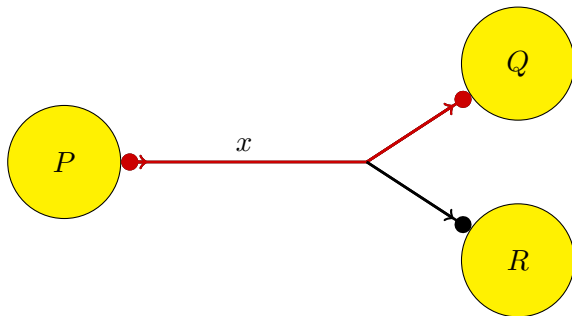
$$\bar{x}.P \mid x.Q \rightarrow P \mid Q$$

“ P (sender) and Q (receiver)
can communicate”

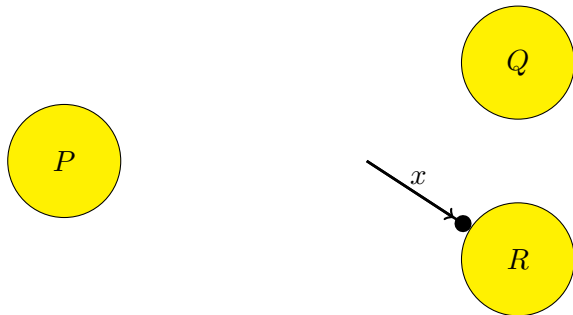
“ P sent a message to Q ”



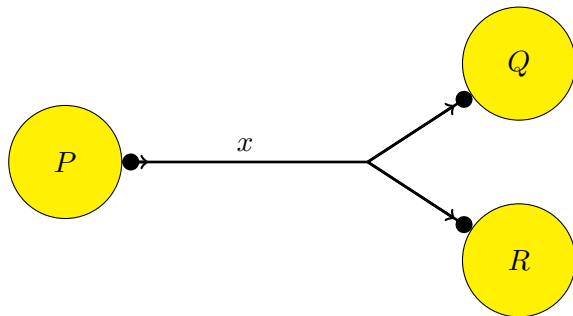
$$\bar{x}.P \mid x.Q \mid x.R$$



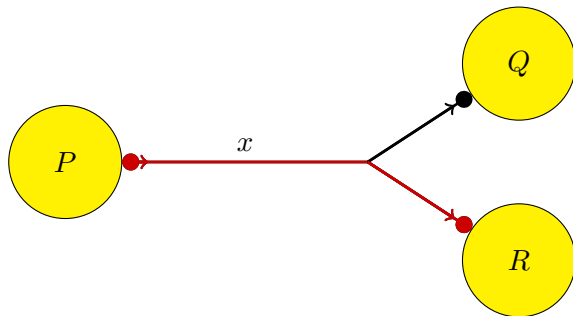
$$\bar{x}.P \mid x.Q \mid x.R$$



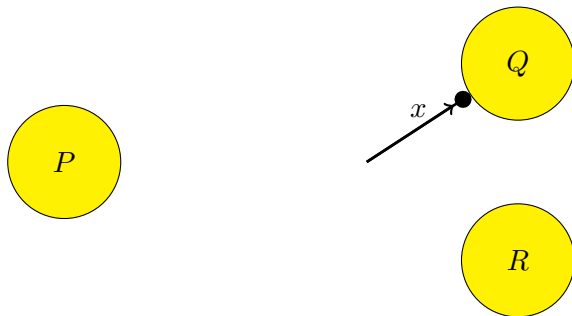
$$\bar{x}.P \mid x.Q \mid x.R \longrightarrow P \mid Q \mid x.R$$



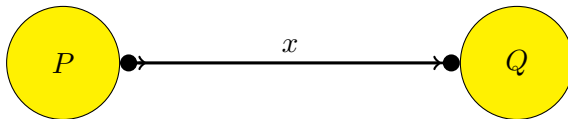
$$\bar{x}.P \mid x.Q \mid x.R \longrightarrow P \mid Q \mid x.R$$



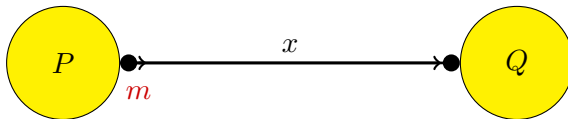
$$\bar{x}.P \mid x.Q \mid x.R \longrightarrow P \mid Q \mid x.R$$



$$\bar{x}.P \mid x.Q \mid x.R \begin{cases} \rightarrow P \mid Q \mid x.R \\ \rightarrow P \mid x.Q \mid R \end{cases}$$

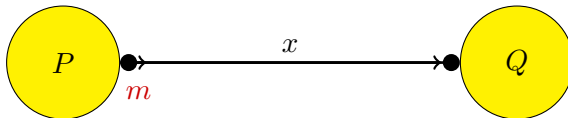


$$\bar{x}.P \mid x.Q \rightarrow P \mid Q$$



$$\bar{x}m.P \mid x.Q \quad \rightarrow \quad P \mid Q$$

“ P sends message m along x ”



$$\bar{x}m.P \mid x(y).Q \rightarrow P \mid Q$$

“ P sends message m along x ”

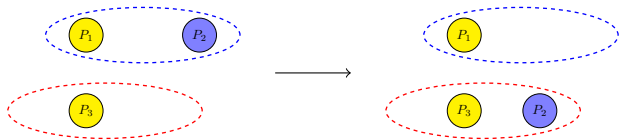


$$\bar{x}m.P \mid x(y).Q \rightarrow P \mid Q[m/y]$$

“ P sends message m along x ”

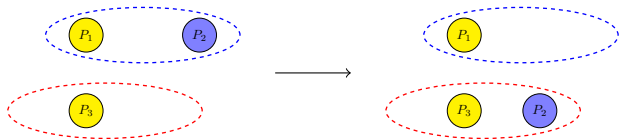
Mobility: Approaches

1. **Processes change** their location in the **physical space** of the processes

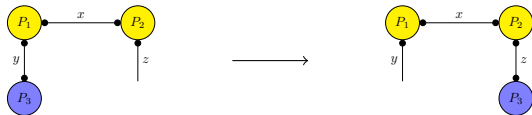


Mobility: Approaches

1. Processes change their location in the **physical space** of the processes

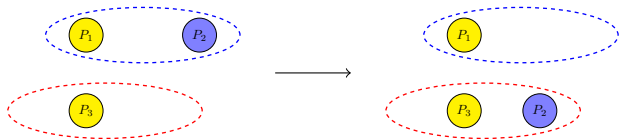


2. Processes change their location in the **virtual space** of **connected** processes

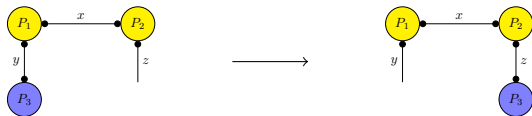


Mobility: Approaches

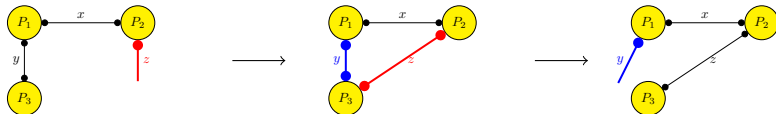
1. Processes change their location in the **physical space** of the processes



2. Processes change their location in the **virtual space** of **connected** processes



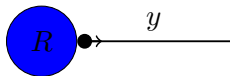
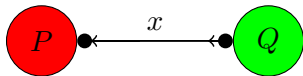
3. **Connections** change their location in the **virtual space** of the **connected** processes (approach of the π -calculus, contains the second approach)



Mobility (2)

How are connections moved?

⇒ Send them as a message!

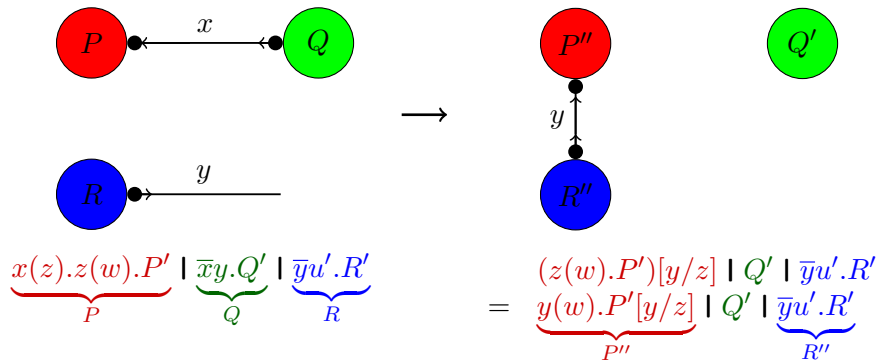


$$\underbrace{x(z).z(w).P'}_P \mid \underbrace{\bar{x}y.Q'}_Q \mid \underbrace{\bar{y}u'.R'}_R$$

Mobility (2)

How are connections moved?

⇒ Send them as a message!

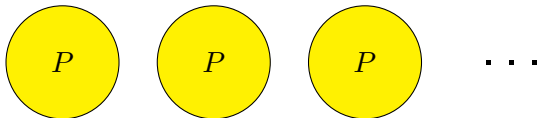


$$\nu x.P$$

“Channel x is private for P ”

Example: $\nu x. (x(y).P \mid \bar{x}z.Q) \mid \bar{x}z'.R$

- No communication between R and P allowed
- equivalent to $\nu x'. (x'(y).P \mid \bar{x}'z.Q) \mid \bar{x}z'.R$



$!P$

“ $!P$ means $\underbrace{P \mid P \mid P \mid \dots}_{\text{infinitely many parallel copies of } P}$ ”

Synchronous π -Calculus Without Sums and With Replication

Syntax

- $\mathcal{N}$ countably infinite set of **names** (similar to variables)
- Syntax for π -calculus-**processes** ($x \in \mathcal{N}$)

P	$::=$	$\pi.P$	(action)
		$P_1 \mid P_2$	(parallel composition)
		$!P$	(replication)
		0	(Inactive process)
		$\nu x.P$	(Restriction)

- Syntax for **action prefixes** where $x, y \in \mathcal{N}$

π	$::=$	$x(y)$	Input
		$\bar{x}y$	Output

Synchronous π -Calculus (2)

- $x(y).P$ means:
 - Receive a name via the **channel** named x and bind it to y .
 - After the name has been received:
Behave like P , where the received name is used for y .

- $x(y).P$ means:
 - Receive a name via the **channel** named x and bind it to y .
 - After the name has been received:
Behave like P , where the received name is used for y .
- $\bar{x}y.P$ means:
 - Send the name y via the **channel** named x .
 - Then behave like P .

- $x(y).P$ means:
 - Receive a name via the **channel** named x and bind it to y .
 - After the name has been received:
Behave like P , where the received name is used for y .
- $\bar{x}y.P$ means:
 - Send the name y via the **channel** named x .
 - Then behave like P .
- 0 is the inactive process that does nothing.

- $x(y).P$ means:
 - Receive a name via the **channel** named x and bind it to y .
 - After the name has been received:
Behave like P , where the received name is used for y .
- $\bar{x}y.P$ means:
 - Send the name y via the **channel** named x .
 - Then behave like P .
- 0 is the inactive process that does nothing.
- $P_1 \mid P_2$ is the parallel composition of P_1 and P_2 .

- $x(y).P$ means:
 - Receive a name via the **channel** named x and bind it to y .
 - After the name has been received:
Behave like P , where the received name is used for y .
- $\bar{x}y.P$ means:
 - Send the name y via the **channel** named x .
 - Then behave like P .
- 0 is the inactive process that does nothing.
- $P_1 \mid P_2$ is the parallel composition of P_1 and P_2 .
- $\nu x.P$ introduces P as **scope** for x .

- $x(y).P$ means:
 - Receive a name via the **channel** named x and bind it to y .
 - After the name has been received:
Behave like P , where the received name is used for y .
- $\bar{x}y.P$ means:
 - Send the name y via the **channel** named x .
 - Then behave like P .
- 0 is the inactive process that does nothing.
- $P_1 \mid P_2$ is the parallel composition of P_1 and P_2 .
- $\nu x.P$ introduces P as **scope** for x .
- $!P$ stands for an infinite number of parallel executions of P , i.e. $\underbrace{P \mid P \mid \dots \mid P}_{\text{infinitely often}}$.

Binding Scopes

Binders in the π -calculus are:

- Restriction $\nu z.P$ binds the name z with scope P
- $x(y).P$ binds y with scope P
- Note: The output prefix $\bar{x}y$ binds neither x nor y .

Free names $\text{fn}(P)$

$$\text{fn}(x(y).P) = \{x\} \cup (\text{fn}(P) \setminus \{y\})$$

$$\text{fn}(\bar{x}y.P) = \{x, y\} \cup \text{fn}(P)$$

$$\text{fn}(P_1 \mid P_2) = \text{fn}(P_1) \cup \text{fn}(P_2)$$

$$\text{fn}(\mathbf{0}) = \emptyset$$

$$\text{fn}(\nu x.P) = \text{fn}(P) \setminus \{x\}$$

$$\text{fn}(!P) = \text{fn}(P)$$

Bound names $\text{bn}(P)$

$$\text{bn}(x(y).P) = \{y\} \cup \text{bn}(P)$$

$$\text{bn}(\bar{x}y.P) = \text{bn}(P)$$

$$\text{bn}(P_1 \mid P_2) = \text{bn}(P_1) \cup \text{bn}(P_2)$$

$$\text{bn}(\mathbf{0}) = \emptyset$$

$$\text{bn}(\nu x.P) = \{x\} \cup \text{bn}(P)$$

$$\text{bn}(!P) = \text{bn}(P)$$

All names of a process: $\text{n}(P) := \text{fn}(P) \cup \text{bn}(P)$

Process context D

- Process that has a hole $[\cdot]$ at a position instead of a process.
- Grammar:

$$D ::= [\cdot] \mid \pi.D \mid D \mid P \mid P \mid D \mid !D \mid \nu x.D \quad \text{for } x \in \mathcal{N}$$

Alpha renaming step

There are two ways to rename bound names:

$$\begin{aligned} D[x(y).P] &\xrightarrow{\alpha} D[x(z).P[z/y]] \text{ if } z \notin n(x(y).P) \\ D[\nu y.P] &\xrightarrow{\alpha} D[\nu z.P'[z/y]] \text{ if } z \notin n(\nu y.P) \end{aligned}$$

Alpha-Equivalence

- $=_{\alpha}$ is the **reflexive-transitive closure** of $\xrightarrow{\alpha}$.
- Processes P_1, P_2 are called **α -equivalent** if $P_1 =_{\alpha} P_2$ holds.

As before:

- Alpha-equivalent processes are considered equal.
- Assumption: Distinct Name Convention

Structural Congruence

Structural Congruence $\equiv$: which processes are considered “structurally equivalent”.
Formally: $\equiv$ is the smallest congruence on processes that satisfies the following axioms.

$$\begin{aligned}P &\equiv Q, \text{ if } P \text{ and } Q \text{ are } \alpha\text{-equivalent} \\P_1 \mid (P_2 \mid P_3) &\equiv (P_1 \mid P_2) \mid P_3 \\P_1 \mid P_2 &\equiv P_2 \mid P_1 \\P \mid \mathbf{0} &\equiv P \\\nu z. \nu w. P &\equiv \nu w. \nu z. P \\\nu z. \mathbf{0} &\equiv \mathbf{0} \\\nu z. (P_1 \mid P_2) &\equiv P_1 \mid \nu z. P_2, \text{ if } z \notin \text{fn}(P_1) \\!P &\equiv P \mid !P\end{aligned}$$

Structural Congruence $\equiv$: which processes are considered “structurally equivalent”.
Formally: $\equiv$ is the smallest congruence on processes that satisfies the following axioms.

$$\begin{aligned} P &\equiv Q, \text{ if } P \text{ and } Q \text{ are } \alpha\text{-equivalent} \\ P_1 \mid (P_2 \mid P_3) &\equiv (P_1 \mid P_2) \mid P_3 \\ P_1 \mid P_2 &\equiv P_2 \mid P_1 \\ P \mid \mathbf{0} &\equiv P \\ \nu z. \nu w. P &\equiv \nu w. \nu z. P \\ \nu z. \mathbf{0} &\equiv \mathbf{0} \\ \nu z. (P_1 \mid P_2) &\equiv P_1 \mid \nu z. P_2, \text{ if } z \notin \text{fn}(P_1) \\ !P &\equiv P \mid !P \end{aligned}$$

- Conversely, if P_1 is transformed into P_2 using the axioms (applied to subprocesses) and α -renaming, then P_1 and P_2 are structurally congruent (i.e., $P_1 \equiv P_2$).

- Structural congruence is actually intended to regard processes as equal because they have more or less the same structure.
- In fact, it has not yet been proven that $\equiv$ is even **decidable**:
Decision problem: Given P_1, P_2 . Does $P_1 \equiv P_2$ hold?
- Known: $\equiv$ is EXPSPACE-hard! (see literature)
- EXPSPACE-hardness already comes from $!$ and $|$
- Together with ν , it apparently becomes even more difficult.
- There are decidable variants.

Reduction rules

(INTERACT)	$x(y).P \mid \bar{x}v.Q \rightarrow P[v/y] \mid Q$
(PAR)	$P \mid Q \rightarrow P' \mid Q$, if $P \rightarrow P'$
(NEW)	$\nu x.P \rightarrow \nu x.P'$, if $P \rightarrow P'$
(STRUCTCONGR)	$P \rightarrow P'$, if $Q \rightarrow Q'$, $P \equiv Q$ and $P' \equiv Q'$

Alternative definition:

- Let the set $\mathcal{R}$ of reduction contexts be described by $R ::= [\cdot] \mid R \mid P \mid \nu x.R$
- Reduction $\rightarrow$:
If $R \in \mathcal{R}$, $P_0 \equiv R[x(y).P \mid \bar{x}v.Q]$, $Q_0 \equiv R[P[v/y] \mid Q]$, then $P_0 \rightarrow Q_0$.

Examples

Consider: $P \equiv \underbrace{x(y).\bar{y}y.\mathbf{0}}_{P_1} \mid \underbrace{\bar{x}z.\mathbf{0}}_{P_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3}$

Consider: $P \equiv \underbrace{x(y).\bar{y}y.\mathbf{0}}_{P_1} \mid \underbrace{\bar{x}z.\mathbf{0}}_{P_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3}$

- P_1 and P_2 communicate via channel x , whereby P_2 sends the name z to P_1 :

$$P \rightarrow \underbrace{\bar{z}z.\mathbf{0}}_{P'_1} \mid \underbrace{\mathbf{0}}_{P'_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3} \equiv P'$$

Consider: $P \equiv \underbrace{x(y).\bar{y}y.\mathbf{0}}_{P_1} \mid \underbrace{\bar{x}z.\mathbf{0}}_{P_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3}$

- P_1 and P_2 communicate via channel x , whereby P_2 sends the name z to P_1 :

$$P \rightarrow \underbrace{\bar{z}z.\mathbf{0}}_{P'_1} \mid \underbrace{\mathbf{0}}_{P'_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3} \equiv P'$$

- P'_1 and P_3 communicate via channel z , whereby P'_1 sends the name z :

$$P' \rightarrow \mathbf{0} \mid \mathbf{0} \mid \mathbf{0} \equiv P''$$

Consider: $P \equiv \underbrace{x(y).\bar{y}y.\mathbf{0}}_{P_1} \mid \underbrace{\bar{x}z.\mathbf{0}}_{P_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3}$

- P_1 and P_2 communicate via channel x , whereby P_2 sends the name z to P_1 :

$$P \rightarrow \underbrace{\bar{z}z.\mathbf{0}}_{P'_1} \mid \underbrace{\mathbf{0}}_{P'_2} \mid \underbrace{z(w).\mathbf{0}}_{P_3} \equiv P'$$

- P'_1 and P_3 communicate via channel z , whereby P'_1 sends the name z :

$$P' \rightarrow \mathbf{0} \mid \mathbf{0} \mid \mathbf{0} \equiv P''$$

- The result is structurally congruent to $\mathbf{0}$ and cannot be reduced further.

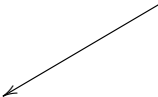
Examples (2)

- The reduction is **non-deterministic**
- Consider:

$$P \equiv x(y).0 \mid \bar{x}v.0 \mid x(z).\bar{z}w.0$$

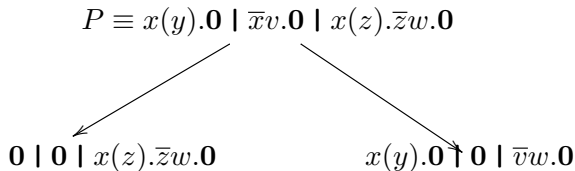
Examples (2)

- The reduction is **non-deterministic**
- Consider:

$$P \equiv x(y).0 \mid \bar{x}v.0 \mid x(z).\bar{z}w.0$$

$$0 \mid 0 \mid x(z).\bar{z}w.0$$

Examples (2)

- The reduction is **non-deterministic**
- Consider:



Examples (3)

- ν -binders prevent communication possibilities and enable local communication that is protected from external interference.

Examples (3)

- ν -binders prevent communication possibilities and enable local communication that is protected from external interference.
- Consider:

$$P \equiv \nu x. (x(y).0 \mid \bar{x}v.0) \mid x(z).\bar{z}w.0$$

Examples (3)

- ν -binders prevent communication possibilities and enable local communication that is protected from external interference.
- Consider:

$$\begin{aligned} P &\equiv \nu x. (x(y).\mathbf{0} \mid \bar{x}v.\mathbf{0}) \mid x(z).\bar{z}w.\mathbf{0} \\ &\rightarrow \nu x. (\mathbf{0} \mid \mathbf{0}) \mid x(z).\bar{z}w.\mathbf{0} \end{aligned}$$

- Only one reduction possible!

Examples (3)

- ν -binders prevent communication possibilities and enable local communication that is protected from external interference.
- Consider:

$$\begin{aligned} P &\equiv \nu x. (x(y).0 \mid \bar{x}v.0) \mid x(z).\bar{z}w.0 \\ &\rightarrow \nu x. (0 \mid 0) \mid x(z).\bar{z}w.0 \end{aligned}$$

- Only one reduction possible!
- After α -renaming of P :

$$P \equiv \nu x'. (x'(y).0 \mid \bar{x'}v.0) \mid x(z).\bar{z}w.0$$

- Local names can leave their binding scope through communication.

- Local names can leave their binding scope through communication.
- Consider

$$P \equiv x(y).P_1 \mid \nu z.\bar{x}z.P_2$$

where z does not appear in P_1 .

- Local names can leave their binding scope through communication.
- Consider

$$P \equiv x(y).P_1 \mid \nu z.\bar{x}z.P_2$$

where z does not appear in P_1 .

- When reducing with (INTERACT):

$$P \equiv \nu z.(x(y).P_1 \mid \bar{x}z.P_2) \rightarrow \nu z.(P_1[z/y] \mid P_2)$$

- The name z , which appears to be known only for P_2 , also becomes known for P_1 .
- This phenomenon is referred to as **extrusion**.

Infinite Reduction Sequences

- Infinitely long reduction sequences are possible
- Example:

$$P = !(\bar{x}v.0) \mid ! (x(z).0)$$

Infinite Reduction Sequences

- Infinitely long reduction sequences are possible
- Example:

$$P = !(\bar{x}v.0) \mid ! (x(z).0)$$

- Start of reduction

Infinite Reduction Sequences

- Infinitely long reduction sequences are possible
- Example:

$$P = !(\bar{x}v.0) \mid ! (x(z).0)$$

- Start of reduction

$$P \quad = \quad !(\bar{x}v.0) \mid ! (x(z).0)$$

Infinite Reduction Sequences

- Infinitely long reduction sequences are possible
- Example:

$$P = !(\bar{x}v.0) \mid ! (x(z).0)$$

- Start of reduction

$$\begin{aligned} P &= !(\bar{x}v.0) \mid ! (x(z).0) \\ &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid ! (x(z).0) \end{aligned}$$

Infinite Reduction Sequences

- Infinitely long reduction sequences are possible
- Example:

$$P = !(\bar{x}v.0) \mid ! (x(z).0)$$

- Start of reduction

$$\begin{aligned} P &= !(\bar{x}v.0) \mid ! (x(z).0) \\ &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid ! (x(z).0) \\ &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid x(z).0 \mid ! (x(z).0) \end{aligned}$$

Infinite Reduction Sequences

- Infinitely long reduction sequences are possible
- Example:

$$P = !(\bar{x}v.0) \mid ! (x(z).0)$$

- Start of reduction

$$\begin{aligned}
 P &= !(\bar{x}v.0) \mid ! (x(z).0) \\
 &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid ! (x(z).0) \\
 &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid x(z).0 \mid ! (x(z).0) \\
 &\rightarrow !(\bar{x}v.0) \mid 0 \mid 0 \mid ! (x(z).0)
 \end{aligned}$$

Infinite Reduction Sequences

- Infinitely long reduction sequences are possible
- Example:

$$P = !(\bar{x}v.0) \mid ! (x(z).0)$$

- Start of reduction

$$\begin{aligned}
 P &= !(\bar{x}v.0) \mid ! (x(z).0) \\
 &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid ! (x(z).0) \\
 &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid x(z).0 \mid ! (x(z).0) \\
 &\rightarrow !(\bar{x}v.0) \mid 0 \mid 0 \mid ! (x(z).0) \\
 &\equiv !(\bar{x}v.0) \mid ! (x(z).0)
 \end{aligned}$$

Infinite Reduction Sequences

- Infinitely long reduction sequences are possible
- Example:

$$P = !(\bar{x}v.0) \mid ! (x(z).0)$$

- Start of reduction

$$\begin{aligned}
 P &= !(\bar{x}v.0) \mid ! (x(z).0) \\
 &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid ! (x(z).0) \\
 &\equiv !(\bar{x}v.0) \mid \bar{x}v.0 \mid x(z).0 \mid ! (x(z).0) \\
 &\rightarrow !(\bar{x}v.0) \mid 0 \mid 0 \mid ! (x(z).0) \\
 &\equiv !(\bar{x}v.0) \mid ! (x(z).0) \\
 &\equiv P
 \end{aligned}$$

- Proved by Robin Milner, 1992
- Encodes the call-by-name lambda calculus into the π -calculus
- Translation $\llbracket \cdot \rrbracket$
- For lambda expression s : $\llbracket s \rrbracket u$ is a process
- where u is a name
- s gets its arguments via u

Translation of abstractions

$$\llbracket \lambda x.s \rrbracket u := u(x).u(v).\llbracket s \rrbracket v$$

- on channel u , the abstraction receives two names:
 - the name x ,
 - the name v : the channel where s receives its arguments

Translation of abstractions

$$\llbracket \lambda x.s \rrbracket u := u(x).u(v).\llbracket s \rrbracket v$$

- on channel u , the abstraction receives two names:
 - the name x ,
 - the name v : the channel where s receives its arguments

Translation of variables

$$\llbracket x \rrbracket u := \bar{x}u.0$$

Encoding $\llbracket \cdot \rrbracket$

Translation of abstractions

$$\llbracket \lambda x.s \rrbracket u := u(x).u(v).\llbracket s \rrbracket v$$

- on channel u , the abstraction receives two names:
 - the name x ,
 - the name v : the channel where s receives its arguments

Translation of variables

$$\llbracket x \rrbracket u := \bar{x}u.\mathbf{0}$$

Example: $\llbracket \lambda x.x \rrbracket u = u(x).u(v).\bar{x}v.\mathbf{0}$

Translation of applications

- In $(s \ t)$, a binding $x = t$ is created for the argument as a separate process.
- The process for $x = t$ requests a channel name each time: on this channel it receives its arguments

- Abbreviation:

$$\langle x = t \rangle := ! (x(w). \llbracket t \rrbracket w)$$

- Replication: Since t may be used more often
- Translation of the application, where v is a new name:

$$\llbracket s \ t \rrbracket u := \nu v. (\llbracket s \rrbracket v \mid \nu x. \bar{v}x. \bar{v}u. \langle x = t \rangle)$$

Example

$$\llbracket (\lambda x.x) \ t \rrbracket u$$

Example

$$\llbracket (\lambda x.x) \ t \rrbracket u \quad = \quad \nu v. (\llbracket (\lambda x.x) \rrbracket v \mid \nu y. \bar{v}y. \bar{v}u. \langle y = t \rangle)$$

Example

$$\begin{aligned}\llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle)\end{aligned}$$

Example

$$\begin{aligned}\llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle)\end{aligned}$$

Example

$$\begin{aligned}\llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\equiv \nu y.\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \bar{v}y.\bar{v}u.\langle y = t \rangle)\end{aligned}$$

Example

$$\begin{aligned} \llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\equiv \nu y.\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(v(w).\bar{y}w.\mathbf{0} \mid \bar{v}u.\langle y = t \rangle) \end{aligned}$$

Example

$$\begin{aligned}\llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\equiv \nu y.\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(v(w).\bar{y}w.\mathbf{0} \mid \bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid \langle y = t \rangle)\end{aligned}$$

Example

$$\begin{aligned}\llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\equiv \nu y.\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(v(w).\bar{y}w.\mathbf{0} \mid \bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid \langle y = t \rangle) \\ &= \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid !(y(w).\llbracket t \rrbracket w))\end{aligned}$$

Example

$$\begin{aligned} \llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\equiv \nu y.\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(v(w).\bar{y}w.\mathbf{0} \mid \bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid \langle y = t \rangle) \\ &= \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid !(y(w).\llbracket t \rrbracket w)) \\ &\equiv \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid (y(w).\llbracket t \rrbracket w) \mid !(y(w).\llbracket t \rrbracket w)) \end{aligned}$$

Example

$$\begin{aligned} \llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\equiv \nu y.\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(v(w).\bar{y}w.\mathbf{0} \mid \bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid \langle y = t \rangle) \\ &= \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid !(y(w).\llbracket t \rrbracket w)) \\ &\equiv \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid (y(w).\llbracket t \rrbracket w) \mid !(y(w).\llbracket t \rrbracket w)) \\ &\rightarrow \nu y.\nu v.(\mathbf{0} \mid (\llbracket t \rrbracket u) \mid !(y(w).\llbracket t \rrbracket w)) \end{aligned}$$

Example

$$\begin{aligned} \llbracket (\lambda x.x) \ t \rrbracket u &= \nu v.(\llbracket (\lambda x.x) \rrbracket v \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\llbracket x \rrbracket w \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &= \nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \nu y.\bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\equiv \nu y.\nu v.(v(x).v(w).\bar{x}w.\mathbf{0} \mid \bar{v}y.\bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(v(w).\bar{y}w.\mathbf{0} \mid \bar{v}u.\langle y = t \rangle) \\ &\rightarrow \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid \langle y = t \rangle) \\ &= \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid !(y(w).\llbracket t \rrbracket w)) \\ &\equiv \nu y.\nu v.(\bar{y}u.\mathbf{0} \mid (y(w).\llbracket t \rrbracket w) \mid !(y(w).\llbracket t \rrbracket w)) \\ &\rightarrow \nu y.\nu v.(\mathbf{0} \mid (\llbracket t \rrbracket u) \mid !(y(w).\llbracket t \rrbracket w)) \\ &\equiv \llbracket t \rrbracket u \mid \nu y.(!(y(w).\llbracket t \rrbracket w)) \end{aligned}$$

Example (2)

In summary:

$$\llbracket (\lambda x.x) \ t \rrbracket u \xrightarrow{*} \llbracket t \rrbracket u \mid \nu y.(! (y(w). \llbracket t \rrbracket w))$$

- $\nu y.(! (y(w). \llbracket t \rrbracket w))$ cannot communicate, therefore “garbage”
- In call-by-name lambda calculus: $(\lambda x.x) \ t \rightarrow t$

Proposition

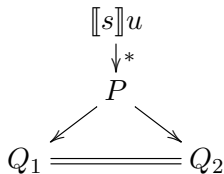
Let s be a closed lambda expression, then one of the following conditions holds:

- 1 $s \downarrow_{name}$ and $\llbracket s \rrbracket u \xrightarrow{*} P$, such that P is irreducible.
- 2 $s \uparrow_{name}$ and there is no P such that $\llbracket s \rrbracket u \xrightarrow{*} P$, where P is irreducible.

Informal: Call-by-name evaluation of lambda expressions can be simulated in the π -calculus.

Lemma

Let s be a closed lambda expression, then $\llbracket s \rrbracket u$ is *deterministic with respect to $\equiv$* , i.e. if $\llbracket s \rrbracket u \xrightarrow{*} P$ and $P \rightarrow Q_1$ as well as $P \rightarrow Q_2$, then $Q_1 \equiv Q_2$ always holds.



Overall, it follows that:

Theorem

The synchronous π -calculus is Turing-complete.

- The result only shows that lambda expressions can be executed.
- No connection between equality theories.
- Milner 1992 shows stronger statements about $\llbracket \cdot \rrbracket$.
- However, we have not (yet) defined a concept of equality for the π -calculus
- There are different concepts!

- So far: Communication takes place in one step, i.e., **synchronous** communication
- In the **asynchronous** π -calculus: Almost everything is identical, with only one difference:
- After the output prefix $\bar{x}y$, only the inactive process $\mathbf{0}$ may follow.

- So far: Communication takes place in one step, i.e., **synchronous** communication
- In the **asynchronous** π -calculus: Almost everything is identical, with only one difference:
- After the output prefix $\bar{x}y$, only the inactive process $\mathbf{0}$ may follow.

Syntax

$$P ::= \mathbf{0} \mid \bar{x}y.\mathbf{0} \mid x(y).P \mid P_1 \mid P_2 \mid \nu z.P \mid !P$$

The Asynchronous π -Calculus (2)

- The asynchronous character comes from the perspective that:
- $\overline{x}y.0$ are **sent** messages that “float around” in space

The Asynchronous π -Calculus (2)

- The asynchronous character comes from the perspective that:
- $\overline{x}y.0$ are **sent** messages that “float around” in space

$$P \equiv \underbrace{\overline{x_1}y_1.0 \mid \dots \mid \overline{x_n}y_n.0}_{\text{medium of the sent messages (buffer)}} \mid \underbrace{Q_1 \mid \dots \mid Q_n}_{\text{“real” program}}$$

- The converse is clear: The asynchronous π -calculus is a subcalculus of the synchronous π -calculus.

- The converse is clear: The asynchronous π -calculus is a subcalculus of the synchronous π -calculus.
- Let $P' \mid Q' \equiv (\bar{x}z.P \mid x(y).Q)$ be a process of the synchronous π -calculus.
- We call P' the **sender**, Q' the **receiver**
- Goal: P and Q are blocked until communication has taken place.

- The converse is clear: The asynchronous π -calculus is a subcalculus of the synchronous π -calculus.
- Let $P' \mid Q' \equiv (\bar{x}z.P \mid x(y).Q)$ be a process of the synchronous π -calculus.
- We call P' the **sender**, Q' the **receiver**
- Goal: P and Q are blocked until communication has taken place.
- First idea: Translate $\bar{x}z.P$ as $\bar{x}z.0 \mid P$
- Does not work: P can continue to reduce without waiting for communication!

- The converse is clear: The asynchronous π -calculus is a subcalculus of the synchronous π -calculus.
- Let $P' \mid Q' \equiv (\bar{x}z.P \mid x(y).Q)$ be a process of the synchronous π -calculus.
- We call P' the **sender**, Q' the **receiver**
- Goal: P and Q are blocked until communication has taken place.
- First idea: Translate $\bar{x}z.P$ as $\bar{x}z.0 \mid P$
- Does not work: P can continue to reduce without waiting for communication!
- Idea: Use asynchronous communication, with the sender waiting for a confirmation of receipt.

Sender $S = \bar{x}z.\mathbf{0} \mid u(v).P$
Receiver $E = x(y).(\bar{u}v.\mathbf{0} \mid Q)$

Evaluation:

$$\begin{aligned} & S \mid E \\ = & (\bar{x}z.\mathbf{0} \mid u(v).P) \mid x(y).(\bar{u}v.\mathbf{0} \mid Q) \\ \rightarrow & u(v).P \mid \bar{u}v.\mathbf{0} \mid Q[z/y] \\ \rightarrow & P \mid Q[z/y] \end{aligned}$$

- Weakness: Not context-independent:
- Translation of $D[P' \mid Q']$ does not ensure that other processes from D communicate via channel u .
- Example $D = u(w).0 \mid [\cdot]$
- Then $D[S \mid E]$ can be reduced as follows:

$$\begin{aligned} & u(w).0 \mid (\bar{x}z.0 \mid u(v).P) \mid x(y).(\bar{u}v.0 \mid Q) \\ \rightarrow & u(w).0 \mid (u(v).P) \mid \bar{u}v.0 \mid Q[z/y] \\ \rightarrow & (u(v).P) \mid Q[z/y] \end{aligned}$$

- Solution: first exchange a **private** channel name between sender and receiver

Encoding with Acknowledgment (3)

- Sender $\nu u.(\bar{x}u.\mathbf{0} \mid u(v)P)$
- Then $u(v)P$ can only be further reduced once u has been sent via x , since u was previously **invisible** to the outside world!
- $u(v)P$ is blocked until communication has taken place.
- This allows synchronization.

Encoding with Acknowledgment (4)

Translation $\llbracket \cdot \rrbracket_\pi$ translates synchronous processes into asynchronous processes

$$\llbracket \mathbf{0} \rrbracket_\pi = \mathbf{0}$$

$$\begin{aligned} \llbracket \bar{x}y.P \rrbracket_\pi &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}y.\mathbf{0} \mid \llbracket P \rrbracket_\pi)), \\ &\text{where } u, v \notin \text{fn}(P) \end{aligned}$$

$$\llbracket x(y).P \rrbracket_\pi = x(u).\nu v.(\bar{u}v.\mathbf{0} \mid v(y).\llbracket P \rrbracket_\pi), \text{ where } u, v \notin \text{fn}(P)$$

$$\llbracket P \mid Q \rrbracket_\pi = \llbracket P \rrbracket_\pi \mid \llbracket Q \rrbracket_\pi$$

$$\llbracket !P \rrbracket_\pi = !\llbracket P \rrbracket_\pi$$

$$\llbracket \nu x.P \rrbracket_\pi = \nu x.\llbracket P \rrbracket_\pi$$

Example

Evaluation of $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$$

Example

Evaluation of $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\begin{aligned} & \llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi \\ &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid x(u').\nu v'.(\bar{u'}v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \end{aligned}$$

Example

Evaluation of $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\begin{aligned} & \llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi \\ &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid x(u').\nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\ &\rightarrow \nu u.(\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi) \mid \nu v'.(\bar{u}v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \end{aligned}$$

Example

Evaluation of $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\begin{aligned} & \llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi \\ &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid x(u').\nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\ &\rightarrow \nu u.(\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi) \mid \nu v'.(\bar{u}v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\ &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid (\bar{v}'z.\mathbf{0}) \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \end{aligned}$$

Evaluation of $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\begin{aligned}
 & \llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi \\
 &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid x(u').\nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\
 &\rightarrow \nu u.(\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi) \mid \nu v'.(\bar{u}v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\
 &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid (\bar{v}'z.\mathbf{0}) \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\
 &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid \mathbf{0} \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid \llbracket Q \rrbracket_\pi[z/y]))
 \end{aligned}$$

Evaluation of $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\begin{aligned}
 & \llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi \\
 &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid x(u').\nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\
 &\rightarrow \nu u.(\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi) \mid \nu v'.(\bar{u}v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\
 &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid (\bar{v}'z.\mathbf{0}) \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\
 &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid \mathbf{0} \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid \llbracket Q \rrbracket_\pi[z/y])) \\
 &\equiv \llbracket P \rrbracket_\pi \mid \llbracket Q \rrbracket_\pi[z/y]
 \end{aligned}$$

Evaluation of $\llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi$:

$$\begin{aligned}
 & \llbracket \bar{x}z.P \mid x(y).Q \rrbracket_\pi \\
 &= \nu u.(\bar{x}u.\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi)) \mid x(u').\nu v'.(\bar{u}'v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi) \\
 &\rightarrow \nu u.(\mathbf{0} \mid u(v).(\bar{v}z.\mathbf{0} \mid \llbracket P \rrbracket_\pi) \mid \nu v'.(\bar{u}v'.\mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\
 &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid (\bar{v}'z.\mathbf{0}) \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid v'(y).\llbracket Q \rrbracket_\pi)) \\
 &\rightarrow \nu v'.(\nu u.(\mathbf{0} \mid \mathbf{0} \mid \llbracket P \rrbracket_\pi \mid \mathbf{0} \mid \llbracket Q \rrbracket_\pi[z/y])) \\
 &\equiv \llbracket P \rrbracket_\pi \mid \llbracket Q \rrbracket_\pi[z/y]
 \end{aligned}$$

- P can continue computing before z has been sent.
- But communication is safe; only Q knows v' .

Notation for output-closed processes

Definition

An **output-closed** process of the synchronous or asynchronous π -calculus is a process such that for every output prefix $\bar{x}y$: The names x and y are bound by binders.

Notation for input-responses

Definition

A process P is an **input-response** if $P \equiv \nu x_1 \dots \nu x_n. x(y). P' \mid Q$

Result for convergence:

Theorem

Let P be an output-closed process of the synchronous π -calculus. Then the following applies:

$$\begin{aligned} &\exists \text{ input-response } P': P \xrightarrow{*} P' \\ &\quad \text{if and only if} \\ &\exists \text{ input-response } P'': \llbracket P \rrbracket_{\pi} \xrightarrow{*} P'' \end{aligned}$$

- Boudol shows even more: Adequacy with regard to contextual preorders

- Synchronous π -calculus **with sums** cannot be (meaningfully) encoded in the asynchronous π -calculus without sums

Requirements:

- Translation is **compositional** and $\llbracket P \mid Q \rrbracket = \llbracket P \rrbracket \mid \llbracket Q \rrbracket$
- Translation is well-behaved with respect to renaming: $\llbracket \sigma(P) \rrbracket = \sigma(\llbracket P \rrbracket)$.

- $P + Q$, behaves like P or Q
- Often restricted: “guarded choice”: Communication must take place during selection, $\pi_1.P + \pi_2.Q$
- We: **Synchronous π -calculus with sums**
- Additional prefix: τ = unobservable action.
- We use the summation symbol Σ and $+$

$$\pi ::= x(y) \mid \bar{x}y \mid \tau \quad \text{where } x, y \in \mathcal{N}$$

$$P ::= \pi.P \mid P_1 \mid P_2 \mid !P \mid \mathbf{0} \mid \nu x.P \mid \sum_i \pi.P_i \quad \text{for } x \in \mathcal{N}$$

- Structural congruence, new: $P + Q \equiv Q + P$,
 $((P_1 + P_2) + P_3) \equiv (P_1 + (P_2 + P_3))$, $P + P \equiv P$.

(SILENT)	$\tau.P \rightarrow P$
(SILENTSUM)	$\tau.P + M \rightarrow P$
(INTERACTSUM)	$x(y).P + M \mid \bar{x}v.Q + N \rightarrow P[v/y] \mid Q$
(INTERACT)	$x(y).P \mid \bar{x}v.Q \rightarrow P[v/y] \mid Q$
(PAR)	$P \mid Q \rightarrow P' \mid Q, \text{ if } P \rightarrow P'$
(NEW)	$\nu x.P \rightarrow \nu x.P', \text{ if } P \rightarrow P'$
(STRUCTCONGR)	$P \rightarrow P', \text{ if } Q \rightarrow Q' \text{ and } P \equiv Q \text{ and } P' \equiv Q'$

Encoding of Non-Deterministic Choice

Unguarded coice:

$$\llbracket P + Q \rrbracket = \nu l. (\bar{l}l. \mathbf{0} \mid l(-).P \mid l(-).Q)$$

here $_$ means a don't care name and the sent name l is irrelevant

Input-Guarded Choice:

$$\llbracket a(x).P + b(y).Q \rrbracket = \nu l. (\bar{l}l. \mathbf{0} \mid a(x).l(-).P \mid b(y).l(-).Q)$$

There are more complex encodings that throw back the received message for the process that cannot receive l

Mixed Choice $a(x).P + \bar{b}y.Q$ is not encodable, the counter example is similar to:

$$a(x).P_1 + \bar{b}c.P_2 \mid b(y).Q_1 + \bar{a}d.Q_2$$

Polyadic π -Calculus

- Previously: **monadic** π -calculus: Only one name is communicated

Polyadic π -Calculus

- Previously: **monadic** π -calculus: Only one name is communicated
- **polyadic** π -calculus: Tuples of names are exchanged
- The action prefix π in the **polyadic** π -calculus:

$$\pi ::= x(\overrightarrow{y}) \mid \overline{x} \overrightarrow{z} \text{ where } x, y_i, z_i \in \mathcal{N}$$

Polyadic π -Calculus

- Previously: **monadic** π -calculus: Only one name is communicated
- polyadic** π -calculus: Tuples of names are exchanged
- The action prefix π in the **polyadic** π -calculus:

$$\pi ::= x(\vec{y}) \mid \bar{x}\vec{z} \text{ where } x, y_i, z_i \in \mathcal{N}$$

- INTERACT rule:

$$(\text{INTERACT}) \quad x(\vec{y}).P \mid \bar{x}\vec{z}.Q \rightarrow P[\vec{z}/\vec{y}] \mid Q \text{ if } |\vec{y}| = |\vec{z}|$$

- $P[\vec{z}/\vec{y}]$ means $P[z_1/y_1, \dots, z_n/y_n]$ if $|\vec{z}| = n$

- Previously: **monadic** π -calculus: Only one name is communicated
- **polyadic** π -calculus: Tuples of names are exchanged
- The action prefix π in the **polyadic** π -calculus:

$$\pi ::= x(\vec{y}) \mid \bar{x}\vec{z} \text{ where } x, y_i, z_i \in \mathcal{N}$$

- INTERACT rule:

$$(\text{INTERACT}) \quad x(\vec{y}).P \mid \bar{x}\vec{z}.Q \rightarrow P[\vec{z}/\vec{y}] \mid Q \text{ if } |\vec{y}| = |\vec{z}|$$

- $P[\vec{z}/\vec{y}]$ means $P[z_1/y_1, \dots, z_n/y_n]$ if $|\vec{z}| = n$
- Prohibited: Using the same channel with different tuple lengths, e.g.,
 $x(a, b).P \mid x(a, b, c).Q$
- Formally via sorts (like a type system)

Translation: Polyadic π -Calculus in Monadic

- Idea: Send elements from tuple $(z_1, \dots, z_n)$ one after the other
- Translate $x(y_1, \dots, y_n).P$ as $x(y_1).\dots x(y_n).P'$
- and translate $\bar{x}(z_1, \dots, z_n).Q$ as $\bar{x}z_1.\dots \bar{x}z_n.Q'$

Translation: Polyadic π -Calculus in Monadic

- Idea: Send elements from tuple $(z_1, \dots, z_n)$ one after the other
- Translate $x(y_1, \dots, y_n).P$ as $x(y_1).\dots x(y_n).P'$
- and translate $\bar{x}(z_1, \dots, z_n).Q$ as $\bar{x}z_1.\dots \bar{x}z_n.Q'$

Does not work, $P := \bar{x}(z_1, z_2).0 \mid \bar{x}(z_3, z_4).0 \mid x(y_1, y_2).\bar{y}_1y_2.0$

Different possibilities to reduce P :

- $P \rightarrow 0 \mid \bar{x}(z_3, z_4).0 \mid \bar{z}_1z_2.0$
- $P \rightarrow \bar{x}(z_1, z_2).0 \mid \bar{z}_3z_4.0$

Translation: Polyadic π -Calculus in Monadic

- Idea: Send elements from tuple $(z_1, \dots, z_n)$ one after the other
- Translate $x(y_1, \dots, y_n).P$ as $x(y_1).\dots x(y_n).P'$
- and translate $\bar{x}(z_1, \dots, z_n).Q$ as $\bar{x}z_1.\dots \bar{x}z_n.Q'$

Does not work, $P := \bar{x}(z_1, z_2).0 \mid \bar{x}(z_3, z_4).0 \mid x(y_1, y_2).\bar{y}_1y_2.0$

Different possibilities to reduce P :

- $P \rightarrow 0 \mid \bar{x}(z_3, z_4).0 \mid \bar{z}_1z_2.0$
- $P \rightarrow \bar{x}(z_1, z_2).0 \mid \bar{z}_3z_4.0$

Translation $Q := \bar{x}z_1.\bar{x}z_2.0 \mid \bar{x}z_3.\bar{x}z_4.0 \mid x(y_1).x(y_2).\bar{y}_1y_2.0$

Reductions for Q :

- $Q \xrightarrow{*} 0 \mid \bar{x}z_3.\bar{x}z_4.0 \mid \bar{z}_1z_2.0$
- $Q \xrightarrow{*} \bar{x}z_1.\bar{x}z_2.0 \mid 0 \mid \bar{z}_3z_4.0$
- $Q \xrightarrow{*} \bar{x}z_2.0 \mid \bar{x}z_4.0 \mid \bar{z}_1z_3.0$
- $Q \xrightarrow{*} \bar{x}z_2.0 \mid \bar{x}z_4.0 \mid \bar{z}_3z_1.0$

Translation: Polyadic π -Calculus in Monadic (2)

- Correction: Sender and receiver exchange private names

Correct translation:

$$\llbracket x(z_1, \dots, z_n).P \rrbracket_{poly} = x(w).w(z_1) \dots w(z_n).\llbracket P \rrbracket_{poly}$$

$$\llbracket \bar{x}(z_1, \dots, z_n).P \rrbracket_{poly} = \nu v.\bar{x}v.\bar{v}z_1 \dots \bar{v}z_n.\llbracket P \rrbracket_{poly}$$

$$\llbracket \nu x.P \rrbracket_{poly} = \nu x.\llbracket P \rrbracket_{poly}$$

$$\llbracket P \mid Q \rrbracket_{poly} = \llbracket P \rrbracket_{poly} \mid \llbracket Q \rrbracket_{poly}$$

$$\llbracket !P \rrbracket_{poly} = !\llbracket P \rrbracket_{poly}$$

$$\llbracket 0 \rrbracket_{poly} = 0$$

Recursive Definitions

- Instead of replication: recursive process definitions
- We replace replication with **constant applications** $A(x_1, \dots, x_n)$

$$P := \dots | \textcolor{blue}{\mathcal{P}} | \dots | A(x_1, \dots, x_n) \text{ where } x_i \in \mathcal{N}$$

- Definition of A outside the program

$$A(x_1, \dots, x_n) = P \text{ where } \text{fn}(P) \subseteq \{x_1, \dots, x_n\}$$

- Instead of replication: recursive process definitions
- We replace replication with **constant applications** $A(x_1, \dots, x_n)$

$$P := \dots | \textcolor{blue}{\mathcal{P}} | \dots | A(x_1, \dots, x_n) \text{ where } x_i \in \mathcal{N}$$

- Definition of A outside the program

$$A(x_1, \dots, x_n) = P \text{ where } \text{fn}(P) \subseteq \{x_1, \dots, x_n\}$$

- New reduction rule:

$$\begin{array}{l} (\text{CONST}) \quad A(y_1, \dots, y_n) \rightarrow P[y_1/x_1, \dots, y_n/x_n] \\ \quad \text{if } A(x_1, \dots, x_n) = P \text{ is the definition of } A \end{array}$$

- Let $\{A_1, \dots, A_n\}$ be all defined constants, with definition $A_i(\vec{x}_i) = P_i$.
- New channel names $a_1, \dots, a_n$
- Each constant application $A_i(\vec{y}_i)$ is replaced by $\langle A_i(\vec{y}_i) \rangle := \overline{a_i} \vec{y}_i.0$

- Let $\{A_1, \dots, A_n\}$ be all defined constants, with definition $A_i(\vec{x}_i) = P_i$.
- New channel names $a_1, \dots, a_n$
- Each constant application $A_i(\vec{y}_i)$ is replaced by $\langle A_i(\vec{y}_i) \rangle := \overline{a_i} \vec{y}_i.0$
- Represent definitions of A_i as processes: $A'_i := !a_i(\vec{x}_i).\langle P_i \rangle$.
- Translation:

$$\llbracket Q \rrbracket_{rec} := \nu a_1. \dots \nu a_n. (\langle Q \rangle \mid A'_1 \mid \dots \mid A'_n)$$

- Let $\{A_1, \dots, A_n\}$ be all defined constants, with definition $A_i(\vec{x}_i) = P_i$.
- New channel names $a_1, \dots, a_n$
- Each constant application $A_i(\vec{y}_i)$ is replaced by $\langle A_i(\vec{y}_i) \rangle := \overline{a_i} \vec{y}_i.0$
- Represent definitions of A_i as processes: $A'_i := !a_i(\vec{x}_i).\langle P_i \rangle$.
- Translation:

$$\llbracket Q \rrbracket_{rec} := \nu a_1. \dots \nu a_n. (\langle Q \rangle \mid A'_1 \mid \dots \mid A'_n)$$

- Note: Not compositional, e.g. $\llbracket Q_1 \mid Q_2 \rrbracket_{rec} \neq \llbracket Q_1 \rrbracket_{rec} \mid \llbracket Q_2 \rrbracket_{rec}$.

Translation: Replication in Recursion

- Translation $\llbracket \cdot \rrbracket_{repl}$
- Consider $!P$ with $\{x_1, \dots, x_n\} = \text{fn}(P)$.
- Choose new constant A_P and translate:

$$\llbracket !P \rrbracket_{repl} := A_P(x_1, \dots, x_n)$$

- All other constructs are translated homomorphically.
- Definition for the constant A_P :

$$A_P(x_1, \dots, x_n) = \llbracket P \rrbracket_{repl} \mid A_P(x_1, \dots, x_n)$$

Example

$$Q = !\bar{x}z.0 \mid x(y).0 \mid x(w).0$$

Evaluation of Q : $Q \equiv !\bar{x}z.0 \mid \bar{x}z.0 \mid x(y).0 \mid x(w).0$
 $\rightarrow !\bar{x}z.0 \mid 0 \mid 0 \mid x(w).0 \equiv !\bar{x}z.0 \mid \bar{x}z.0 \mid x(w).0$
 $\rightarrow !\bar{x}z.0 \mid 0 \mid 0 \equiv !\bar{x}z.0$

$$Q = !\bar{x}z.0 \mid x(y).0 \mid x(w).0$$

Evaluation of Q :

$$\begin{aligned} Q &\equiv !\bar{x}z.0 \mid \bar{x}z.0 \mid x(y).0 \mid x(w).0 \\ &\rightarrow !\bar{x}z.0 \mid 0 \mid 0 \mid x(w).0 \equiv !\bar{x}z.0 \mid \bar{x}z.0 \mid x(w).0 \\ &\rightarrow !\bar{x}z.0 \mid 0 \mid 0 \equiv !\bar{x}z.0 \end{aligned}$$

Translation: $\llbracket Q \rrbracket_{repl} = A(x, z) \mid x(y).0 \mid x(w).0$
 where $A(x, z) = \bar{x}z \mid A(x, z)$

An evaluation of $\llbracket Q \rrbracket_{repl}$:

$$\begin{aligned} \llbracket Q \rrbracket_{repl} &= A(x, z) \mid x(y).0 \mid x(w).0 \\ &\rightarrow (\bar{x}z \mid A(x, z)) \mid x(y).0 \mid x(w).0 \\ &\rightarrow A(x, z) \mid 0 \mid x(w).0 \\ &\rightarrow (\bar{x}z \mid A(x, z)) \mid 0 \mid x(w).0 \\ &\rightarrow A(x, z) \mid 0 \mid 0 \end{aligned}$$

- Different notions in the π -calculus, no uniform notion
- No **notion for termination**
- Often: termination of processes is not central, since distributed systems run in infinite loops and thus do not terminate.

- Different notions in the π -calculus, no uniform notion
- No **notion for termination**
- Often: termination of processes is not central, since distributed systems run in infinite loops and thus do not terminate.
- The common equality notion: Processes have the same input/output possibilities
- Instead of reduction, **labeled transition system**
- Labels represent the inputs and outputs of the process
- Distinction between **internal communication** and **external communication**

- External communication perspective: How can a process communicate if a communication partner is added?
- Process of the form $P_0 \equiv \nu z_1. \dots \nu z_n. x(y).P$ with $x, y \neq z_i$ can **receive an input**
- Process of the form $P_0 \equiv \nu z_1. \dots \nu z_n. \bar{x}y.P$ with $x, y \neq z_i$ can **perform an output**
- Other variant: Output of a bound name, i.e., for $P_0 = \nu z_1. \dots \nu z_n. \nu y. \bar{x}y.P$ with $x, y \neq z_i$:

For communication, $\nu y.$ must be shifted

- Labels: $\alpha := \tau \mid x(y) \mid \bar{x}y \mid \bar{x}(y)$
- $\text{fn}(x(y)) = \{x, y\}$ $\text{bn}(x(y)) = \emptyset$
 $\text{fn}(\bar{x}y) = \{x, y\}$ $\text{bn}(\bar{x}y) = \emptyset$
 $\text{fn}(\bar{x}(y)) = \{x\}$ $\text{bn}(\bar{x}(y)) = \{y\}$
 $\text{fn}(\tau) = \emptyset$ $\text{bn}(\tau) = \emptyset$

Labeled Transition System (2)

For synchronous π -calculus:

$$(\text{IN}) \quad x(y).P \xrightarrow{x(z)} P[z/y]$$

$$(\text{OUT}) \quad \bar{x}y.P \xrightarrow{\bar{x}y} P$$

$$(\text{OPEN}) \quad \nu y.P \xrightarrow{\bar{x}(y)} Q, \text{ if } P \xrightarrow{\bar{x}y} Q \text{ and } x \neq y.$$

$$(\text{INTERACT}) \quad x(y).P \mid \bar{x}v.Q \xrightarrow{\tau} P[v/y] \mid Q$$

$$(\text{PAR}) \quad P \mid Q \xrightarrow{\alpha} P' \mid Q, \text{ if } P \xrightarrow{\alpha} P' \text{ and } n(\alpha) \cap \text{fn}(Q) = \emptyset$$

$$(\text{NEW}) \quad \nu x.P \xrightarrow{\alpha} \nu x.P', \text{ if } P \xrightarrow{\alpha} P' \text{ and } x \notin n(\alpha)$$

$$(\text{STRUCTCONGR}) \quad P \xrightarrow{\alpha} P', \text{ if } Q \xrightarrow{\alpha} Q' \text{ and } P \equiv Q \text{ and } P' \equiv Q'$$

- Sums, τ , etc. can be added

Example

$$\begin{array}{l} \nu x. (\overline{x}(u).\overline{u}u.0 \mid \overline{x}w.z(a).\overline{a}b.0) \\ \xrightarrow{\tau} \overline{w}w.0 \mid \underline{z(a)}.\overline{a}b.0 \\ \xrightarrow{z(c)} \overline{w}w.0 \mid \overline{c}b.0 \\ \xrightarrow{\overline{w}w} \overline{c}b.0 \\ \xrightarrow{\overline{c}b} 0 \end{array}$$

Note: The name c can be chosen freely

Harmony Lemma

$P \rightarrow Q$ if and only if $P \xrightarrow{\tau} Q'$ where $Q' \equiv Q$

Definition

A binary relation $\mathcal{R}$ on processes is a **strong bisimulation**, if for all $(P, Q) \in \mathcal{R}$ the following holds

- If $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that the following holds $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{R}$
- If $Q \xrightarrow{\alpha} Q'$, then there exists a process P' such that $P \xrightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{R}$.

Processes P and Q are **strongly bisimilar** (written $P \sim_{b, strong} Q$) if there exists a strong bisimulation R with $(P, Q) \in R$.

The relation $\sim_{b, strong}$ is called **strong bisimilarity**.

Lemma

$\sim_{b, strong}$ is the **largest** strong bisimulation.

- Let $Proc$ be the set of all processes of the π -calculus.

Let $F : (Proc \times Proc) \rightarrow (Proc \times Proc)$ be the following function:

$(P, Q) \in F(\eta)$ if and only if:

- If $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \eta$
 - If $Q \xrightarrow{\alpha} Q'$, then there exists P' such that $P \xrightarrow{\alpha} P'$ and $(P', Q') \in \eta$
- Strong bisimilarity is the **largest fixed point** of the function F
(the largest subset X of $Proc \times Proc$ for which $F(X) = X$ holds)
 - **Principle of coinduction**: Every relation η that is **dense** with respect to F is also contained in the greatest fixed point.
 - A relation η is **F -dense** if $\eta \subseteq F(\eta)$
 - F -dense relation = strong bisimulation

Strong bisimilarity is an equivalence relation, but **not a congruence**:

- Assume $z \neq a$ and
- $P = \bar{z}b.0 \mid a(c).0$
- $Q = \bar{z}b.a(c).0 + a(c).\bar{z}b.0$
- It holds that $P \sim_{b, \text{strong}} Q$:
 $\mathcal{R} = \{(P, Q), (a(c).0, a(c).0), (\bar{z}b.0, \bar{z}b.0), (0, 0)\}$ is a strong bisimulation
- However, $x(z).P \not\sim_{b, \text{strong}} x(z).Q$
- Thus, $P \sim_{b, \text{strong}} Q \implies C[P] \sim_{b, \text{strong}} C[Q]$ does not hold for every context C ,
i.e., $\sim_{b, \text{strong}}$ is not a congruence.

Definition

Two processes P, Q are **strongly fully bisimilar** (notation $P \sim_{b, \text{strong}, \text{full}} Q$) if, for all substitutions σ that replace names with names, $\sigma(P) \sim_{b, \text{strong}} \sigma(Q)$.

- A proof can be provided that $\sim_{b, \text{strong}, \text{full}}$ is a congruence.

Bisimulation

- Neglecting τ -transitions
- $\xRightarrow{\tau} :=$ reflexive-transitive closure of $\xrightarrow{\tau}$
- $\xRightarrow{\alpha} := \xRightarrow{\tau} \xrightarrow{\alpha} \xRightarrow{\tau}$

A binary relation $\mathcal{R}$ on processes is a **bisimulation** if for all $(P, Q) \in \mathcal{R}$ the following holds

- If $P \xrightarrow{\alpha} P'$, then there exists a process Q' such that $Q \xRightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{R}$
- If $Q \xrightarrow{\alpha} Q'$, then there exists a process P' such that $P \xRightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{R}$.

Processes P and Q are **bisimilar** (written $P \sim_b Q$) if there exists a bisimulation $\mathcal{R}$ with $(P, Q) \in \mathcal{R}$. The relation $\sim_b$ is called **bisimilarity**.

Processes P, Q are **fully bisimilar** ($P \sim_{b,full} Q$) if $\sigma(P) \sim_b \sigma(Q)$ holds for all substitutions σ .

- Bisimilarity is not a congruence.
- Full bisimilarity is a congruence.

Theorem

For all processes P, Q , the following holds:

- $P \sim_{b, \text{strong}} Q \implies P \sim_b Q$
- $P \sim_{b, \text{strong}, \text{full}} Q \implies P \sim_{b, \text{full}} Q$.

Furthermore, $\sim_{b, \text{strong}} \neq \sim_b$ and $\sim_{b, \text{strong}, \text{full}} \neq \sim_{b, \text{full}}$.

For process P , we write

- $P \dot{\vdash}^x$, if $P \xrightarrow{x(y)} P'$
- $P \dot{\vdash}^{\bar{x}}$, if $P \xrightarrow{\bar{x}y} P'$ or $P \xrightarrow{\bar{x}(y)} P'$
- for $\beta \in \{x, \bar{x}\}$: $P \downarrow_\beta$ if there is a process Q such that $P \xRightarrow{\tau} Q$ and $Q \dot{\vdash}^\beta$

Lemma

For all processes of the synchronous π -calculus (with sums), the following applies:

- $P \dot{\vdash}^x$ if and only if $P \equiv \nu v_1 \dots \nu v_n. (x(y).P' + M \mid Q)$ where $x \neq v_i$.
- $P \dot{\vdash}^{\bar{x}}$ if and only if $P \equiv \nu v_1 \dots \nu v_n. (\bar{x}y.P' + M \mid Q)$ where $x \neq v_i$.

Therefore: Reduction can also be used instead of labeled transitions!

Definition

A binary relation $\mathcal{R}$ on processes is a **barbed bisimulation** if for all $(P, Q) \in \mathcal{R}$:

- If $P \dot{\vdash}^x$ (or $P \dot{\vdash}^{\bar{x}}$), then $Q \downarrow_x$ (or $Q \downarrow_{\bar{x}}$)
- If $Q \dot{\vdash}^x$ (or $Q \dot{\vdash}^{\bar{x}}$), then $P \downarrow_x$ (or $P \downarrow_{\bar{x}}$)
- If $P \xrightarrow{\tau} P'$, then there exists Q' with $Q \xRightarrow{\tau} Q'$ and $(P', Q') \in \mathcal{R}$
- If $Q \xrightarrow{\tau} Q'$, then there exists P' with $P \xRightarrow{\tau} P'$ and $(P', Q') \in \mathcal{R}$

Two processes P, Q are **barbed bisimilar** (written $P \sim_{b, \text{barbed}} Q$) if there exists a barbed bisimulation containing (P, Q) .

Definition

Two processes P, Q are **barbed congruent** (written $P \sim_{c,b,barbed} Q$) if for all contexts C : $C[P] \sim_{b,barbed} C[Q]$.

Definition

Two processes P, Q are **barbed congruent** (written $P \sim_{c,b,barbed} Q$) if for all contexts C : $C[P] \sim_{b,barbed} C[Q]$.

Theorem

For all processes P, Q , the following holds:

$$P \sim_{b,full} Q \implies P \sim_{c,b,barbed} Q$$

It is unknown whether the converse of this implication holds.

The **may-testing** preorder $\leq_{\text{may}}$:

$P \leq_{\text{may}} Q$ if and only if for all contexts C , for all names x :

$$C[P] \downarrow_x \implies C[Q] \downarrow_x \text{ and } C[P] \downarrow_{\bar{x}} \implies C[Q] \downarrow_{\bar{x}}.$$

May-testing equivalence $\sim_{\text{may}}$:

$P \sim_{\text{may}} Q$ if and only if $P \leq_{\text{may}} Q$ and $Q \leq_{\text{may}} P$

Theorem

The may-testing equivalence is a congruence.

$\sim_{may}$ is very coarse-grained:

Theorem

For all processes P, Q , the following holds: $P \sim_{c,b,barbed} Q \implies P \sim_{may} Q$.

The converse does not hold, e.g. processes with $\not\sim_{c,b,barbed}$ but $\sim_{may}$:

$$a(x).0 \oplus (b(x).0 \oplus c(x).0) \text{ and } (a(x).0 \oplus b(x).0) \oplus c(x).0$$

where $P \oplus Q := \tau.P + \tau.Q$

Too coarse:

$x(y).0 \oplus 0$ and $x(y).0$ are may-testing equivalent!

- $x(y).0 \oplus 0 \xrightarrow{\tau} 0$
- while $x(y).0$ always can receive a name on x

Should-Barb

For $\mu \in \{x, \bar{x}\}$ we write $P \Downarrow_\mu$ (P must have a barb μ) if

$$\forall Q : P \xrightarrow{*} Q \implies Q \Downarrow_\mu .$$

should-testing Preorder $\leq_{\text{should}}$:

$P \leq_{\text{should}} Q$ if and only if for all contexts C , for all names x :

$$C[P] \Downarrow_x \implies C[Q] \Downarrow_x \text{ and } C[P] \Downarrow_{\bar{x}} \implies C[Q] \Downarrow_{\bar{x}} .$$

Should-testing equivalence $\sim_{\text{should}}$:

$P \sim_{\text{should}} Q$ if and only if $P \leq_{\text{should}} Q$ and $Q \leq_{\text{should}} P$

Theorem

For all processes P, Q , the following holds:

$$P \sim_{c,b,barbed} Q \implies P \sim_{should} Q.$$

Theorem

For all processes P, Q , the following applies:

$$P \sim_{should} Q \implies P \sim_{may} Q.$$

Theorem

$$\sim_{b, \text{strong}, \text{full}} \subseteq \sim_{b, \text{full}} \subseteq \sim_{c, b, \text{barbed}} \subseteq \sim_{\text{should}} \subseteq \sim_{\text{may}}$$