π-Calculus

A tutorial and overview

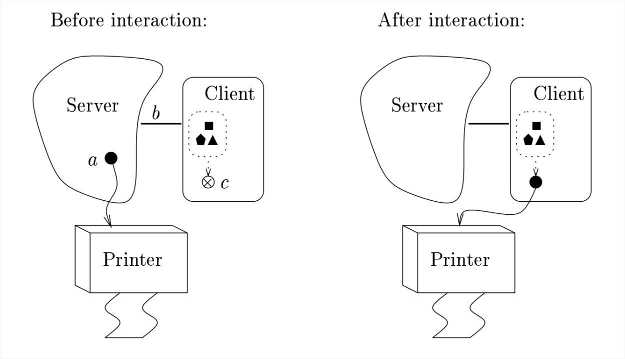
Adam T. Geller
Jonathan W. Lavington

December 20, 2018

University of British Columbia, Vancouver

Introduction

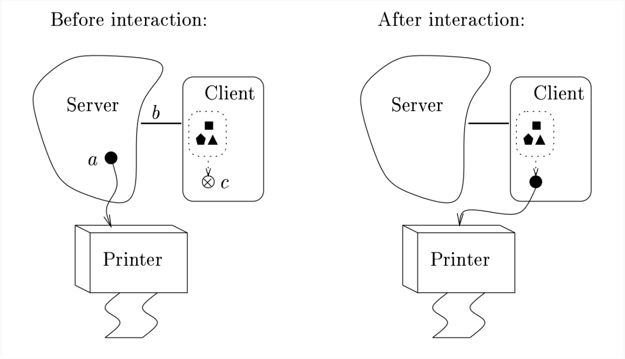
π-Calculus is a mathematical model of processes that change as they interact.



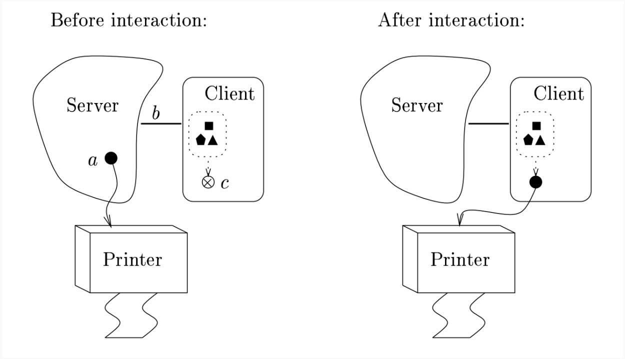
Within this language, the basic step represents movement of a communication link between two or more processes.

Introduction

How can we model this process?



Introduction



If we define the server that sends a along b as the term $\bar{b}a.S$, and the client that receives this link along b as $b(c).\bar{c}d.P$: then under the semantics of π-Calculus this process is described as:

$$\bar{b}a.S|b(c).\bar{c}d.P \xrightarrow{\tau} S|\bar{a}d.P$$

Introduction

What is this really going on?

In the example above, *a*,*b*,*c*, and *d* all represent names associated with access rights.

- *a* represents access to the printer
- *b* accesses the server
- *c* is a placeholder for an access to arrive along *a*
- *d* accesses some data

In the case above we say that the printer “moves” to the client.

π-Calculus Syntax

How do we build a semantics from the ground up to model this process?

π-Calculus Syntax

π-Calculus Syntax

The most basic unit in π-Calculus is a name. There are infinitely many of them and they are assumed to have no structure.
We can denote these using letters such as:

$$x, y, \ldots \in \mathrm{X}$$

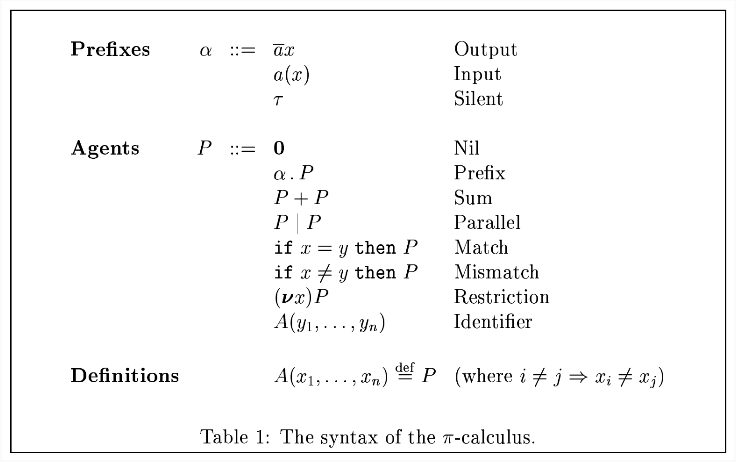
In the simplest version (called Monadic Calculus), there is only one other entity known as a process. We can denote these as:

$$P, Q, \ldots \in \mathrm{P}$$

These processes are just abstractions built from names by the syntax used above.

π-Calculus Syntax

Following the Semantics defined by Parrow, we can build our processes from names under the following syntax:



Prefixes	α	$::=$	$\overline{a}x$	Output
			$a(x)$	Input
			τ	Silent
Agents	P	$::=$	$\mathbf{0}$	Nil
			$\alpha\,.\,P$	Prefix
			$P + P$	Sum
			$P \mid P$	Parallel
			$\texttt{if } x = y \texttt{ then } P$	Match
			$\texttt{if } x \neq y \texttt{ then } P$	Mismatch
			$(\boldsymbol{\nu} x)P$	Restriction
			$A(y_1, \ldots, y_n)$	Identifier
Definitions			$A(x_1, \ldots, x_n) \overset{\text{def}}{=} P$	(where $i \neq j \Rightarrow x_i \neq x_j$)

Table 1: The syntax of the π-calculus.

π-Calculus Syntax

What do all of these symbols mean under the π-Calculus semantics?

Lets start with Prefixes, of which there are two kinds for the Monadic calculus:

1. $a(x)$ (Input Prefix)
   $\rightarrow$ A name is received along a name a, where x is a placeholder for the received name.
2. $\bar{a}x$ (Output Prefix)
   $\rightarrow$ A name x is sent along a name a.

π-Calculus Syntax

These prefixes are then applied using the dot notation. That is, for some prefix α we write an action performed by some expression as $\alpha.P$

For example:

1. $a(x).P$ (Input Prefix)
   $\rightarrow$ A name is received along a name a, where x is a placeholder for the received name. After the input is received, the agent will proceed as P with the new name replacing x.
2. $\bar{a}x.P$ (Output Prefix)
   $\rightarrow$ A name x is sent along a name a, and thereafter the agent continues as P. $\bar{a}$ can be thought of as an output port and x as the datum sent from that port.

π-Calculus Syntax

We are almost there! So how do we define | ?

| represents one of 8 agent definitions under Monadic π-Calculus, and is defined as follows:

3. $P|Q$ (Parrallel Composition)
   Represents the combined actions of both P and Q executing in parallel.

Note the P and Q can be independent, but can also communicate if one performs an output while the other performs an input along the same port.

π-Calculus Syntax

Now that we have everything defined, we can fully express the example from earlier

$$\bar{b}a.S|b(c).\bar{c}d.P \xrightarrow{\tau} S|\bar{a}d.P$$

means that before interaction,

1. The server that sends a along b is defined as $\bar{b}a.S$
2. The client that receives some link along b then sends data along it is given as $b(c).\bar{c}d.P$

and after interaction,

1. The first agent as shown above is now defined as S
2. The second agent now sends data d along along the now transferred link a and is now known as P.

π-Calculus Syntax

We have created a formal representation of the process that was happening above (using a imprecise definition of $\xrightarrow{\tau}$).

Great!

How is π-Calculus different from other process algebras?

The answer is that π-Calculus admits migrating local scopes.

π-Calculus Syntax

Most process algebras can declare a communication link between local processes, known as a restriction.

In π-Calculus this is defined as:

4. $(\nu x).P$ (Restriction)
   The agent behaves as P, but the name x is local, meaning that it can't be immediately used as a port for communication between P and its environment.

Note however, that **it can be used for communication between components of P**.

π-Calculus Syntax

So how is this form of restriction different from similar restrictions in other process algebras?

It's similar in that no other process can use the restricted name immediately.

The difference is that **the restricted name is still a transferable object** that can be sent by its own process to another which would then be allowed to access and use it.

π-Calculus Syntax

If we return to the original example, suppose *a* is a local link between the server *S* and printer *R*.

Then this is captured by:

$$(\nu a)(\bar{b}a.S|R)$$

We can send *a* along *b* to the client, resulting in a shared link between all three processes **that is distinct from any other name in other processes**.

This transition is written as:

$$(\nu a)(\bar{b}a.S|R)|b(c).\bar{c}d.P \xrightarrow{\tau} (\nu a)(S|R|\bar{a}d.P)$$

π-Calculus Syntax

Like other process algebras, π-Calculus includes a concept of Nil, Sum and Parallel. Where Parallel is defined as above, and Nil and Sum are given as,

5. $P + Q$ (Sum)
   Represents an agent that can enact either P or Q.
6. 0 (Nil)
   Represents an empty agent, which cannot perform any actions.

Examples of these will be given below, after we define a few more terms...

π-Calculus Syntax

π-Calculus admits Boolean expressions using the Match and Mismatch prefixes, which are defined:

7. if $x = y$ then P (Match)
   The agent will behave as P if x and y are the same, otherwise it does nothing.
8. if $x \neq y$ then P (Mismatch)
   The agent will behave as P if x and y are the not same, otherwise it does nothing.

These if statements seem restrictive compared to other value passing algebras that admit arbitrary Boolean expressions.

π-Calculus Syntax

Why define the semantics in this way?

Because these are the only tests that can be performed in this language!

We assumed earlier we work strictly with combinations of names, which had no structure or operators, so **the only thing we can do is compare for equality**.

π-Calculus Syntax

This may seem restrictive, but if we nest these statements we can actually achieve more complicated behavior. For example:

1. if $x = y$ then if $u \neq v$ then P (conjunctively)
   If $x = y$ and $u \neq v$ then behave as P
2. if $x = y$ then P + if $u \neq v$ then P (disjunctively)
   If $x = y$ or $u \neq v$ then behave as P
3. if $x = y$ then Q + if $x \neq y$ then P (biconditional)
   If $x = y$ then Q else P

π-Calculus Syntax

Before we continue lets make note of some of the vocabulary used when talking about these interacting processes:

1. We say P **is guarded in** Q if P is a proper sub-term of a Prefix in Q.
2. The input Prefix $a(x)$ is said to **bind** x **in** P, and occurrences of x in P are then called **bound**. In contrast output prefix $\bar{a}x$ does not bind x.
3. Within these prefixes we have a **subject** a, and an **object** x.
4. The object is called **free** in the output Prefix, and **bound** in the input Prefix.
5. Additionally, the restriction operator, $(\nu x).P$, also binds x in P.

π-Calculus Syntax

Question:

Can a prefix form have neither a subject nor an object?

Yes!

π-Calculus Syntax

The silent prefix is defined in the following way:

9. $\tau.P$ (Silent Prefix)
   Represents an agent that can evolve to P without any interaction with the environment.

As a result of its definition, the silent prefix has neither a subject, nor an object.

π-Calculus Syntax

As a Quick note, we can define both bound and unbound in the language as well using the following notation:

1. $bn(P)$ (Bound names):
   set of bound names in a process P.
2. $fn(P)$ (Free names):
   set of free names in a process P.

These definition are important, as they lead us to name substitution within π-Calculus.

π-Calculus Syntax

Within π-Calculus, substitution is defined in the following ways:

1. $\{x/y\}$ (substitution)
   A substitution that maps y to x.
2. $\{x_1, x_2 ... x_n / y_1, y_2 ... y_n\}$ (substitution multiple)
   A substitution that maps n distinct y_i to n distinct x_i.
3. $\sigma(x)$ (substitution function)
   Function σ that ranges over the space of all substitutions.

For example the agent $P\sigma$ is the agent P where all names x in P are replaced by $\sigma(x)$ (using alpha conversion to avoid captures).

π-Calculus Syntax

As a result of this capture avoiding behavior, bound names are re-named such that where ever x is replaced by $\sigma(x)$, the new occurrence of $\sigma(x)$ is free!

For example:

$$(a(x).(\nu b)\bar{x}b.\bar{c}y.0)\{x, b/y, c\}$$

could evolve to,

$$a(z).(\nu d)\bar{z}d.\bar{b}x.0$$

π-Calculus Syntax

This also leads to the final piece of syntax in π-Calculus, known as a Definition:

10. $A(x_1, ..., x_n) = P$, where $i \neq j \Rightarrow x_i \neq x_j$ (Identifier A, Definition P)

    Every identifier A has a definition P, and each of the names are assumed to be pairwise distinct.

    Effectively we have that $A(y_1, ..., y_n)$ acts as P when $y_1, ..., y_n$ are replaced with $x_1, ..., x_n$.

In π-Calculus, a definition can be thought of as a process declaration.

π-Calculus Syntax

As a final note, to make notation a bit simpler, we can define sets of operations regarding the restriction operator, and the sum operator.

1. $P_1 + ... + P_n = \sum_{i=1}^{n} P_i$
2. $(\nu x_1)...(\nu x_n)P = (\nu x_1...x_n)P$

π-Calculus Syntax

Now we can reason about whether or not two agents are doing the same thing, for instance:

$P|Q$ and $Q|P$ both represent a parallel composition of the agents P and Q

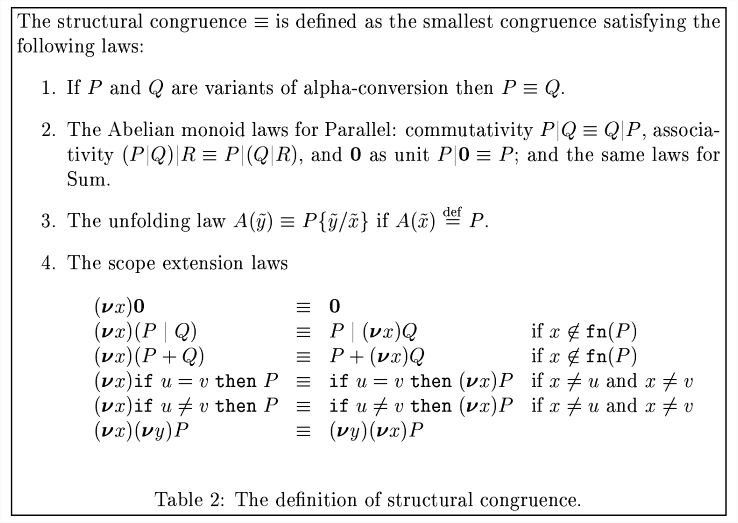
$a(x).\overline{b}x$ and $a(y).\overline{b}y$ only differ in terms of the bound name, and thus intuitively should do the same thing.

So how do we identify agents that are doing the same thing?

Structural Congruence

Structural Congruence

One possible definition of two programs representing the same thing, is called structural congruence.



The structural congruence $\equiv$ is defined as the smallest congruence satisfying the following laws:

1. If P and Q are variants of alpha-conversion then $P \equiv Q$.
2. The Abelian monoid laws for Parallel: commutativity $P|Q \equiv Q|P$, associativity $(P|Q)|R \equiv P|(Q|R)$, and $\mathbf{0}$ as unit $P|\mathbf{0} \equiv P$; and the same laws for Sum.
3. The unfolding law $A(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\}$ if $A(\tilde{x}) \overset{\text{def}}{=} P$.
4. The scope extension laws

$(\boldsymbol{\nu} x)\mathbf{0}$	$\equiv$	$\mathbf{0}$	
$(\boldsymbol{\nu} x)(P \mid Q)$	$\equiv$	$P \mid (\boldsymbol{\nu} x)Q$	if $x \notin \mathtt{fn}(P)$
$(\boldsymbol{\nu} x)(P + Q)$	$\equiv$	$P + (\boldsymbol{\nu} x)Q$	if $x \notin \mathtt{fn}(P)$
$(\boldsymbol{\nu} x)\mathtt{if}\ u = v\ \mathtt{then}\ P$	$\equiv$	$\mathtt{if}\ u = v\ \mathtt{then}\ (\boldsymbol{\nu} x)P$	if $x \neq u$ and $x \neq v$
$(\boldsymbol{\nu} x)\mathtt{if}\ u \neq v\ \mathtt{then}\ P$	$\equiv$	$\mathtt{if}\ u \neq v\ \mathtt{then}\ (\boldsymbol{\nu} x)P$	if $x \neq u$ and $x \neq v$
$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P$	$\equiv$	$(\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P$	

Table 2: The definition of structural congruence.

Structural Congruence

Structural congruence signifies two programs are the same at a local level. That is, it signifies two programs are the same without any notion of operational semantics and transitions.

Structural Congruence

Starting with the first three rules,

1. Alpha conversion signifies **bound variable names don't matter**.
2. Monoid laws indicate **both parallel and sum are inherently un-ordered** and that **because** 0 **is empty, adding it to a process creates no inherent change to the process**.
3. Unfolding law give that **an identifier is the same as its definition with the appropriate parameter instantiation**.

Structural Congruence

The scope laws just indicate that **if a variable doesn’t exist in a component of the expression, a restriction on that component isn’t necessary**.

To see this, consider the first three scope extension laws:

$$\begin{array}{lcll}
(\boldsymbol{\nu} x)\mathbf{0} & \equiv & \mathbf{0} & \\
(\boldsymbol{\nu} x)(P \mid Q) & \equiv & P \mid (\boldsymbol{\nu} x)Q & \text{if } x \notin \mathtt{fn}(P) \\
(\boldsymbol{\nu} x)(P + Q) & \equiv & P + (\boldsymbol{\nu} x)Q & \text{if } x \notin \mathtt{fn}(P)
\end{array}$$

1. A restricted name under an empty context is still empty.
2. If a name doesn’t exist in an sub expression of the parallel composition, then a restriction on it wont affect it, and thus only needs to be applied to sub-expressions that contain it as a free name.
3. Similar to the parallel composition, if a name doesn’t exist in an sub expression of sum, then a restriction on it wont affect it.

Structural Congruence

As a few final notes,

We **do not have**,

$$(\boldsymbol{\nu} x)(P \mid Q) \equiv (\boldsymbol{\nu} x)P \mid (\boldsymbol{\nu} x)Q$$

In the first case, both agents have the same restrictions, but **can use the restricted variables to communicate**.

In the second, both agents only have the same restrictions, and **cannot communicate using the restricted variables**.

π-Calculus Examples

π-Calculus Examples

Now that we have defined structural congruences, we can use it to define operational semantics that reason about **behavioral equivalence** instead of just **syntactic equivalence**.

But before we introduce the operational semantics, we need define some notation and look at a couple examples.

π-Calculus Examples

π-Calculus uses a labeled transition system, such that transitions are of the kind $P \xrightarrow{\alpha} Q$ for some set of actions ranged over by alpha.

1. $\xrightarrow{\tau}$ Internal action
2. $\xrightarrow{a(x)}$ input action of kind $a(x)$
3. $\xrightarrow{\bar{a}x}$ (Free) output action of kind $\bar{a}x$
4. $\xrightarrow{\bar{a}\nu x}$ (Bound) output action of kind $\bar{a}\nu x$

This also means, for a transition $\alpha.P$ there will be a transition α leading to P.

π-Calculus Examples

Now we can start looking at a few examples of scope migration, i.e. how **restrictions move with their objects**.

We will focus specifically on three postulates that facilitate the form of scope migration we are interested in:

1. Parallel components allow for communication between channels $(a(x).P|\bar{a}b.Q \xrightarrow{\tau} P\{b/x\}|Q)$.
2. Silent transitions do not affect restrictions.
3. Structurally congruent agents should not be distinguished, and thus semantics must assign them the same behavior.

π-Calculus Examples

So what are the implications for restricted objects?

π-Calculus Examples

Consider attempting to infer the transition of:

$$a(x).\bar{c}x|(\nu b)(\bar{a}b)$$

First by scope extension, we know this process is structurally congruent to:

$$(\nu b)(a(x).\bar{c}x|\bar{a}b)$$

This agent then has a transition between components given by:

$$a(x).\bar{c}x|\bar{a}b \xrightarrow{\tau} \bar{c}b|0$$

Thus:

$$(\nu b)(a(x).\bar{c}x|\bar{a}b) \xrightarrow{\tau} (\nu b)(\bar{c}b|0)$$

Finally Nil can be omitted by the monoid laws:

$$a(x).\bar{c}x|(\nu b)(\bar{a}b) \xrightarrow{\tau} (\nu b)(\bar{c}b)$$

π-Calculus Examples

As another solidifying example, consider:

$$(\nu b)(a(x).P)|\bar{a}b.Q$$

In this case we cannot immediately extend the scope since b is free on the right side, but we can apply alpha conversion:

$$(\nu b')(a(x).P\{b'/b\})|\bar{a}b.Q$$

Because of the unrestricted access of the right hand side of the parallel composition we can now write the transition as:

$$(\nu b)(a(x).P)|\bar{a}b.Q \xrightarrow{\tau} (\nu b')(P\{b'/b\}\{b/x\})|Q$$

π-Calculus Examples

Through alpha conversion and scope extension:

1. we can send the restricted names between processes
2. the restriction will always move with the name
3. the agents its passed to will never include free occurrences of that name

π-Calculus Examples

As a final example, consider modeling exchange of private resources.

Suppose we have an resource R, that is controlled by a server S which distributes access rights.

In the simplest case, the access right is just to execute R, and can be modeled by:

$$(\nu e)(S|e.R)$$

Here R cannot execute until it receives a signal e. The server can invoke it by calling $\bar{e}$, or it can send e to a client wishing to use R.

π-Calculus Examples

Now suppose a client Q needs the resource.

Here Q asks S along a channel c for the access key e, to the resource R, and upon receipt of this key R can be executed.

This is represented by:

$$c(x).\bar{x}.Q|(\nu e)(\bar{c}e.S|e.R) \xrightarrow{\tau} (\nu e)(\bar{e}.Q|S|e.R) \xrightarrow{\tau} (\nu e)(Q|S|R)$$

In the first transition, Q receives an access to R, and in the second this access is used.

π-Calculus Operational Semantics

π-Calculus Operational Semantics

Now that we have an intuition of how these operations work, onto the operational semantics!

π-Calculus Operational Semantics

Basic points:

1. The restriction operator will not allow an action with a restricted name as a subject.
2. As a result of (1) we also define an additional type of action called a bound output written as $\bar{a}\nu u$, whereby a local name u is transmitted along a, extending the scope of u to the recipient.

π-Calculus Operational Semantics

Back to the different type of actions we can take:

1. internal actions τ
2. free output actions of kind $\bar{a}x$
3. input actions of kind $a(x)$
4. bound output actions of type $\bar{a}\nu x$

Notice that each of these transitions correspond to prefixes in the calculus.

π-Calculus Operational Semantics

With these final definitions in hand, we are ready for the operational semantics!

π-Calculus Operational Semantics

$$\textsc{struct}\quad \frac{P' \equiv P,\ P \xrightarrow{\alpha} Q,\ Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$$

$$\textsc{prefix}\quad \frac{}{\alpha\,.\,P \ \xrightarrow{\alpha}\ P}$$

$$\textsc{sum}\quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\textsc{match}\quad \frac{P \xrightarrow{\alpha} P'}{\texttt{if}\ x = x\ \texttt{then}\ P \xrightarrow{\alpha} P'}$$

$$\textsc{mismatch}\quad \frac{P \xrightarrow{\alpha} P',\ x \neq y}{\texttt{if}\ x \neq y\ \texttt{then}\ P \xrightarrow{\alpha} P'}$$

$$\textsc{par}\quad \frac{P \xrightarrow{\alpha} P',\ \texttt{bn}(\alpha) \cap \texttt{fn}(Q) = \emptyset}{P|Q \xrightarrow{\alpha} P'|Q}$$

$$\textsc{com}\quad \frac{P \xrightarrow{a(x)} P',\ Q \xrightarrow{\overline{a}u} Q'}{P|Q \xrightarrow{\tau} P'\{u/x\}|Q'}$$

$$\textsc{res}\quad \frac{P \xrightarrow{\alpha} P',\ x \notin \alpha}{(\boldsymbol{\nu} x)P \xrightarrow{\alpha} (\boldsymbol{\nu} x)P'}$$

$$\textsc{open}\quad \frac{P \xrightarrow{\overline{a}x} P',\ a \neq x}{(\boldsymbol{\nu} x)P \xrightarrow{\overline{a}\boldsymbol{\nu} x} P'}$$

Go Examples

Go Examples

$$P = (\nu c)\left((\bar{a}x.\bar{b}y.c(r).Q) | (a(n).\bar{c}n + b(m).\bar{c}m)\right)$$

Go Examples

$$
\begin{array}{lr}
P = (\nu c)\left((\bar{a}x.\bar{b}y.c(r).Q)|(a(n).\bar{c}n + b(m).\bar{c}m)\right) & \\
\xrightarrow{\tau} (\nu c)\left((\bar{b}y.c(r).Q)|(\bar{c}x + b(m).\bar{c}m)\right) & \text{(com + struct)} \\
\xrightarrow{\tau} (\nu c)\left((c(r).Q)|(\bar{c}x + \bar{c}y)\right) & \text{(com + struct)} \\
\hline
\text{Now, there's a split:} & \\
(\nu c)\left((c(r).Q)|(\bar{c}x + \bar{c}y)\right) \xrightarrow{\tau} (\nu c)\left((c(r).Q)|\bar{c}x\right) & \text{(sum)} \\
\xrightarrow{\tau} (\nu c)Q\{r/x\} & \text{(com)} \\
\text{but also} & \\
(\nu c)\left((c(r).Q)|(\bar{c}x + \bar{c}y)\right) \xrightarrow{\tau} (\nu c)\left((c(r).Q)|\bar{c}y\right) & \text{(struct + sum)} \\
\xrightarrow{\tau} (\nu c)Q\{r/y\} & \text{(com)}
\end{array}
$$

Go Examples

$$\text{SUM}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$(\bar{c}x + \bar{c}y)$ can enact either $\bar{c}x$ or $\bar{c}y$.
Remember, $P \xrightarrow{\tau} P'$ means that P can immediately enact P'. It doesn't mean that P can *only* enact P'.

Go Examples

$$\text{SUM}\quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$(\bar{c}x + \bar{c}y)$ can enact either $\bar{c}x$ or $\bar{c}y$.
Remember, $P \xrightarrow{\tau} P'$ means that P can immediately enact P'. It doesn't mean that P can *only* enact P'.
A derivation is a proof of possible program behaviour!

Go Examples

Programs can get ”stuck”:

$$(a(b).\bar{c}d.P) \mid (c(e).\bar{a}f.Q)$$

What do we do?
Go’s answer: panic!

For further reading see [1, 2]

Questions?

References i

J. A. Bergstra, A. Ponse, and S. A. Smolka.
Handbook of process algebra.
Elsevier, 2001.

R. Milner.
The polyadic π-calculus: a tutorial.
In *Logic and algebra of specification*, pages 203–246. Springer, 1993.