Advanced Process Calculi

Lecture 1: the pi-calculus

Copenhagen, August 2013

Joachim Parrow

Learning outcomes

After completing the course you will be able to:

- Use modern process calculi to make high-level models.
- Explain key issues involved in their construction, abilities, and limitations.
- Use a prototype tool to analyze your models.

Teachers

- **Joachim Parrow**, professor in Computing Science, Uppsala University
- **Jesper Bengtson**, professor in Computer Science, IT-university Copenhagen
- **Ramunas Gutkovas**, PhD student, Uppsala University

How it works

- 4 **lectures** of 2x45 mins each. Slides will be available after each lecture.
- 4 afternoons of tutored **exercise** and lab sessions.
- **Examination**: individual project. Choose an application and model it. (Examined in September by Jesper Bengtson.)

Material

http://www.it.uu.se/research/group/mobility/apc-course-copenhagen-2013

Advanced Process Calculi

Advanced Process Calculi

- A calculus: something in which we can calculate things.

Advanced Process Calculi

- A calculus: something in which we can calculate things.
- Calculation presupposes a rigorously defined semantics.

Advanced Process Calculi

- A calculus: something in which we can calculate things.
- Calculation presupposes a rigorously defined semantics.
- Calculation = logically obtained conclusion

Advanced Process Calculi

- A calculus: something in which we can calculate things.
- Calculation presupposes a rigorously defined semantics.
- Calculation = logically obtained conclusion
- Note the plural form. There will be more than one...

Advanced Process Calculi

Advanced Process Calculi

- The objects that we calculate with will be processes.

Advanced Process Calculi

- The objects that we calculate with will be processes.
- A process is something that exhibits behaviour through interactions with the environment.

Advanced Process Calculi

- The objects that we calculate with will be processes.
- A process is something that exhibits behaviour through interactions with the environment.
- Defined in an abstract and high-level way. Could be implemented as software or hardware.

Advanced Process Calculi

- The objects that we calculate with will be processes.
- A process is something that exhibits behaviour through interactions with the environment.
- Defined in an abstract and high-level way. Could be implemented as software or hardware.

Eg: ``Send data value 5 along the output channel´´

Advanced Process Calculi

Advanced Process Calculi

- Suggests that there are also basic ones...

Advanced Process Calculi

- Suggests that there are also basic ones...
- No fret! We will start out by recapitulating the pi-calculus, a very basic one. (pi-calculus experts in the audience: see you tomorrow!)

Advanced Process Calculi

- Suggests that there are also basic ones...
- No fret! We will start out by recapitulating the pi-calculus, a very basic one. (pi-calculus experts in the audience: see you tomorrow!)
- Advanced = Complicated? Rich? Powerful? Recent?

The pi-calculus

- Developed in 1987-1992 by Robin Milner, Joachim Parrow and David Walker.
- Goal: give a minimalistic **compositional** computational model encompassing concurrency with **mobility** and **scoping**.

- **minimalistic**: only include stuff necessary to capture concurrency, mobility and scoping

- **minimalistic**: only include stuff necessary to capture concurrency, mobility and scoping
- **concurrency**: asynchronous processes communicate in binary atomic actions

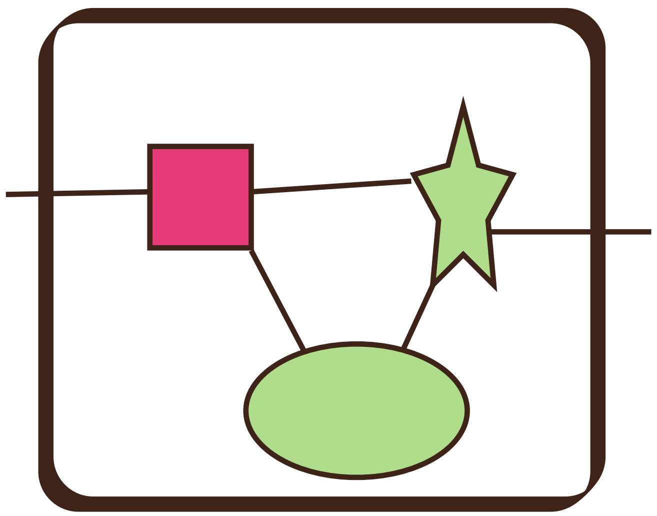
- **minimalistic**: only include stuff necessary to capture concurrency, mobility and scoping
- **concurrency**: asynchronous processes communicate in binary atomic actions
- **mobility**: connections between processes may change during execution

- **minimalistic**: only include stuff necessary to capture concurrency, mobility and scoping
- **concurrency**: asynchronous processes communicate in binary atomic actions
- **mobility**: connections between processes may change during execution
- **scoping**: these conections may be local

- **minimalistic**: only include stuff necessary to capture concurrency, mobility and scoping
- **concurrency**: asynchronous processes communicate in binary atomic actions
- **mobility**: connections between processes may change during execution
- **scoping**: these conections may be local
- Departure: CCS, a process calculus having all of the above except mobility (Milner, 1979 -)

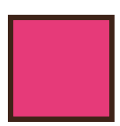
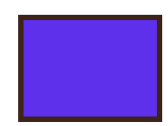
Compositionality

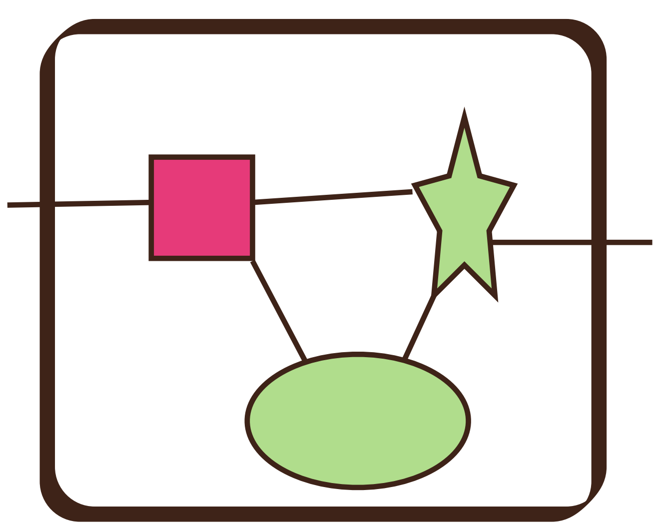
”The behaviour of a system is given by the behaviour of its parts”



Compositionality

”The behaviour of a system is given by the behaviour of its parts”

 behaves as  means that

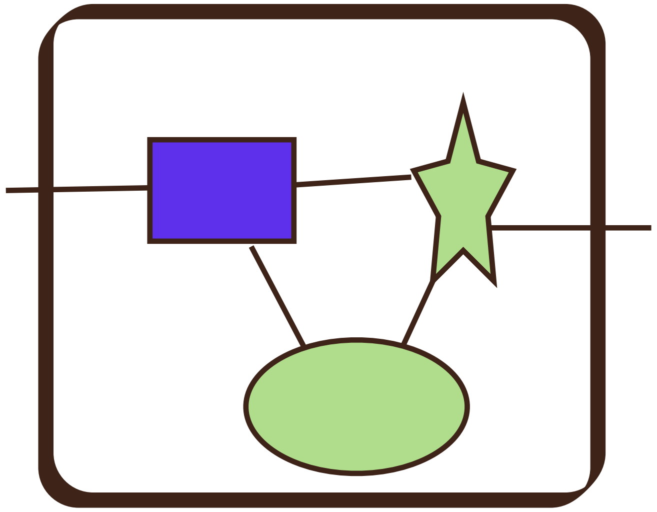


they can replace each other

Compositionality

”The behaviour of a system is given by the behaviour of its parts”

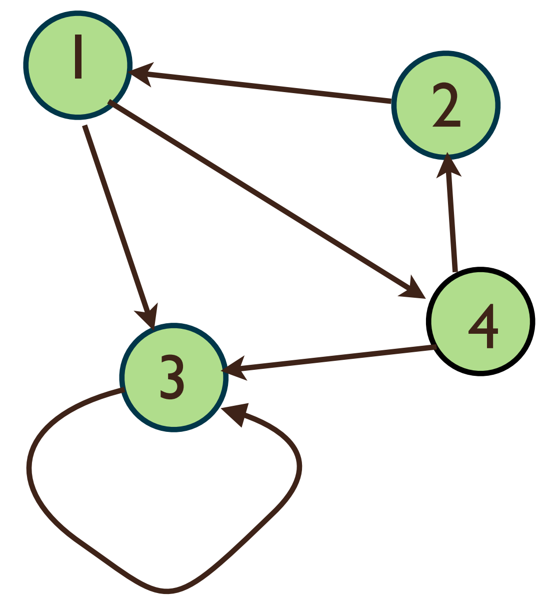
 behaves as  means that



they can replace each other

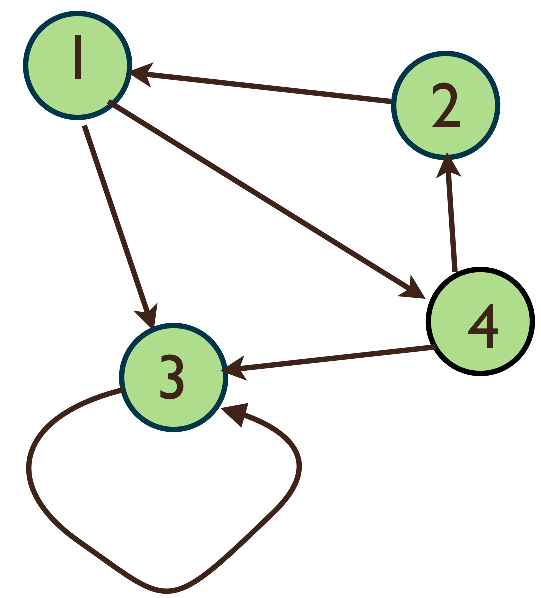
Formally:

If the behaviour $[[A]]$ of a system A is defined as the **transitions** between its **states**

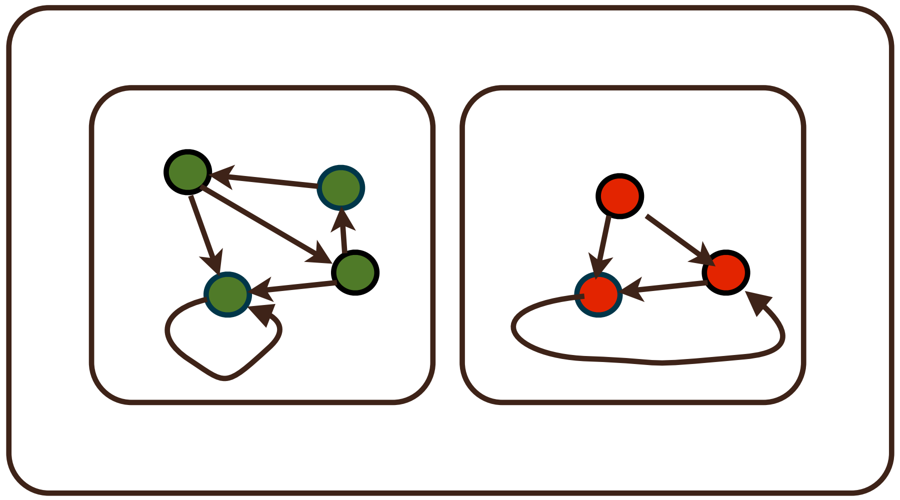


Formally:

If the behaviour $[[A]]$ of a system A is defined as the **transitions** between its **states**

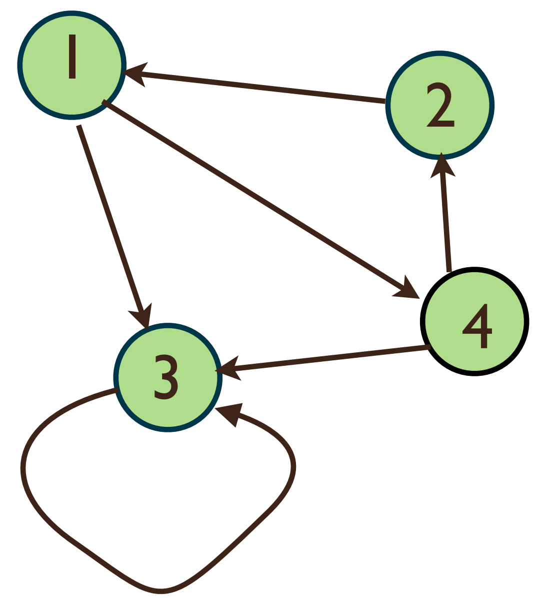


then the states and transitions of a system should be determined by the states and transitions of its **components**.



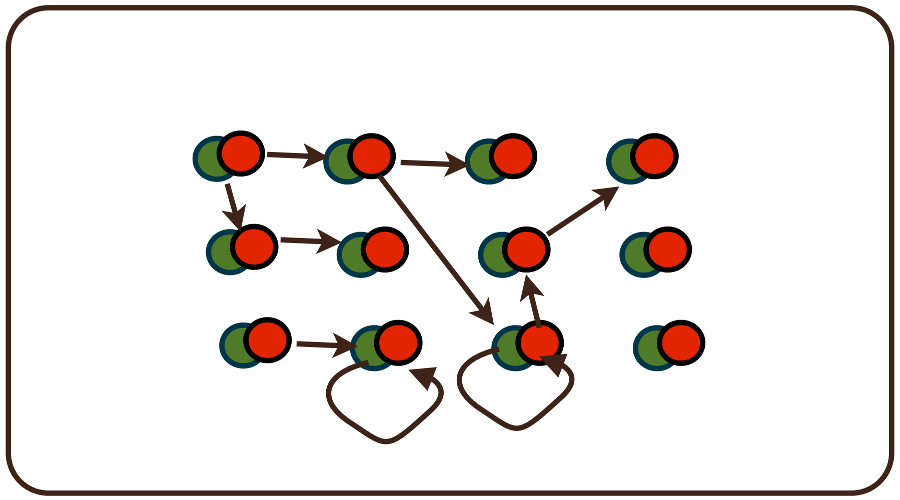
Formally:

If the behaviour $[[A]]$ of a system A is defined as the **transitions** between its **states**



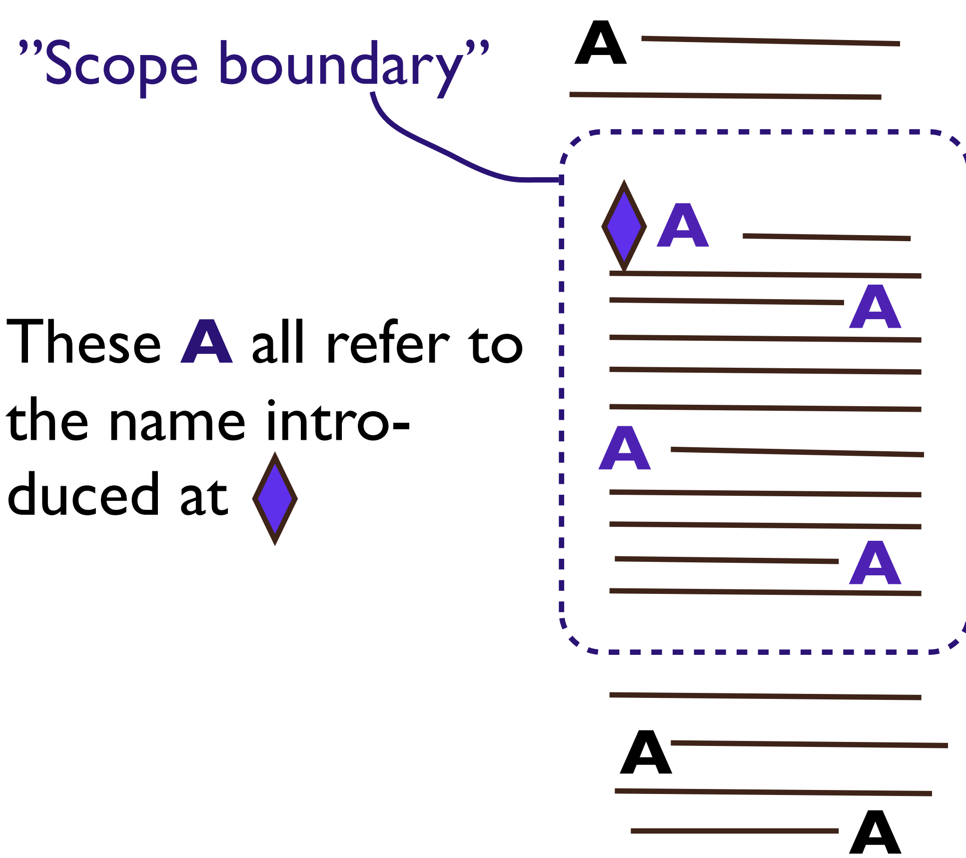
$$[[A \times B]] = [[A]] \times [[B]]$$

then the states and transitions of a system should be determined by the states and transitions of its **components**.



Scoping

When a name is introduced, the valid places of its use, aka scope, is defined.



These **A** all refer to the name intro-duced at ◆

The outer **A** do not refer to that name, even though it is called the same

Scoping

”Declaration of local resource”

Local variable

Scope boundary

```
.
.
k=i+1;
{int i=10;
  while (i>0)
  {i--; k=k+i;};
};
if (i=2)
.
.
```

Scoping

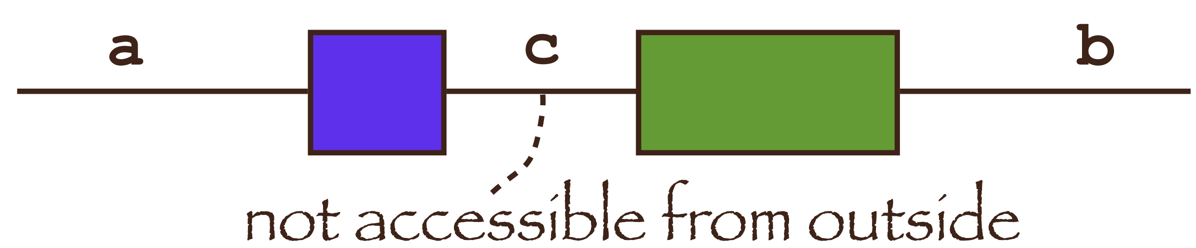
"Declaration of local resource"

Local variable

Scope boundary

```
.
.
k=i+1;
{int i=10;
  while (i>0)
  {i--; k=k+i;};
};
if (i=2)
.
.
```

Local channel



Scoping

"Declaration of local resource"

Local variable

Scope boundary

```
.
.
k=i+1;
{int i=10;
  while (i>0)
  {i--; k=k+i;};
};
if (i=2)
.
.
```

Local channel

Scoping

Universal law #1: Alpha-conversion

A scoped name can systematically be replaced by any other name not already occurring in its scope

```
.
.
k=i+1;
{int i=10;
  while (i>0)
  {i--; k=k+i;};
};
if (i=2)
.
.
```

Scoping

Universal law #1: Alpha-conversion

A scoped name can systematically be replaced by any other name not already occurring in its scope

```
.
.
k=i+1;
{int m=10;
  while (m>0)
  {m--; k=k+m;};
};
if (i=2)
.
.
```

Scoping

Universal law #1: Alpha-conversion

A scoped name can systematically be replaced by any other name not already occurring in its scope

```
.
.
k=i+1;
{int k=10;
  while (k>0)
   {k--; k=k+k;};
};
if (i=2)
.
.
```

Scoping

Universal law #2: Scope extension

A scope can be extended (or retracted) as long as it does not include more (or fewer) occurrences of the scoped name

```
.
.
k=i+1;
{int i=10;
  while (i>0)
  {i--; k=k+i;};
};
k=k*2;
 .
 .
```

Scoping

Universal law #2: Scope extension

A scope can be extended (or retracted) as long as it does not include more (or fewer) occurrences of the scoped name

```
.
.
k=i+1;
{int i=10;
  while (i>0)
  {i--; k=k+i;};
k=k*2;
};
 .
 .
```

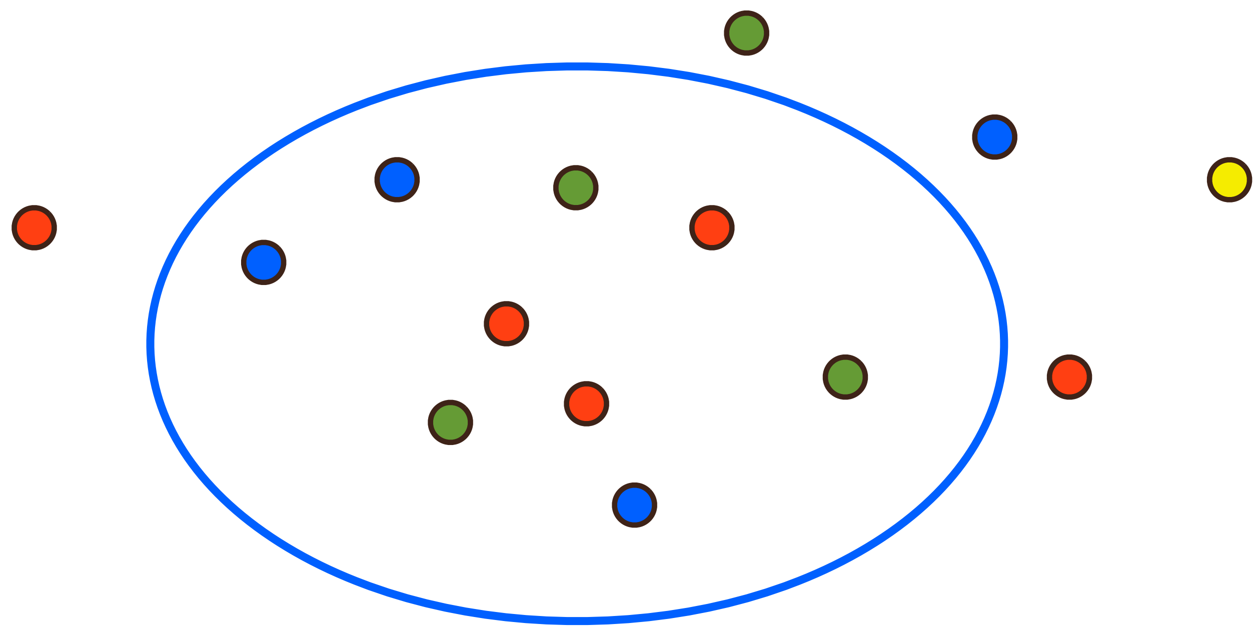
Scoping

Universal law #2: Scope extension

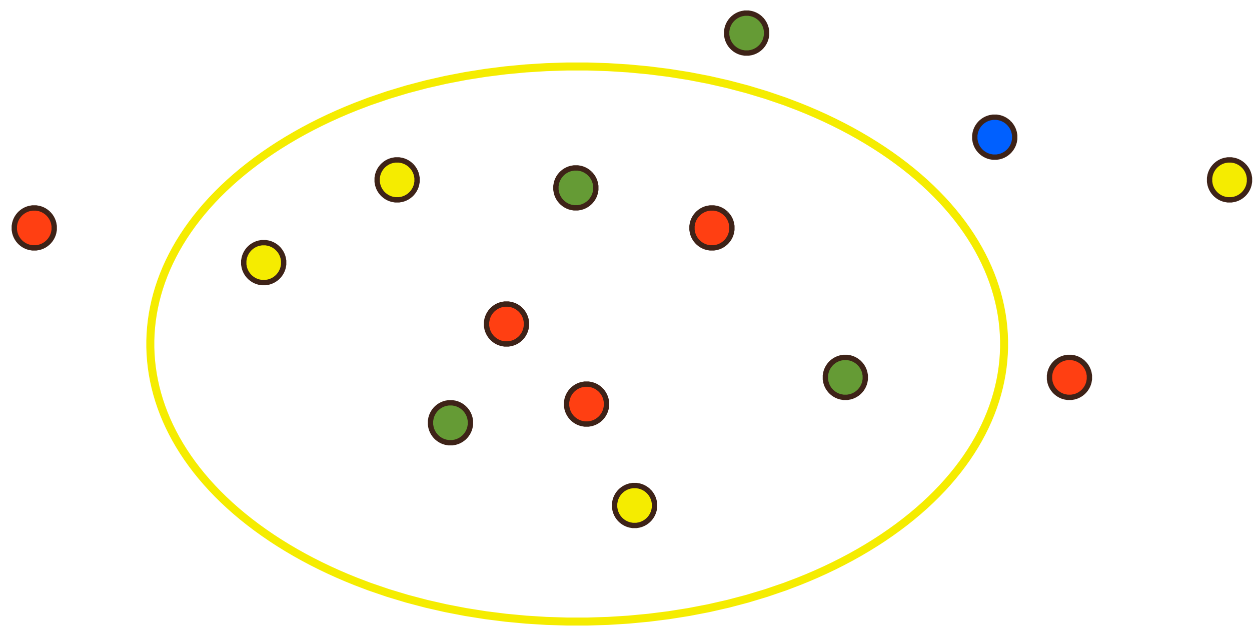
A scope can be extended (or retracted) as long as it does not include more (or fewer) occurrences of the scoped name

```
.
.
{int i=10;
 k=i+1;
  while (i>0)
   {i--; k=k+i;};
k=k*2;
};
 .
 .
```

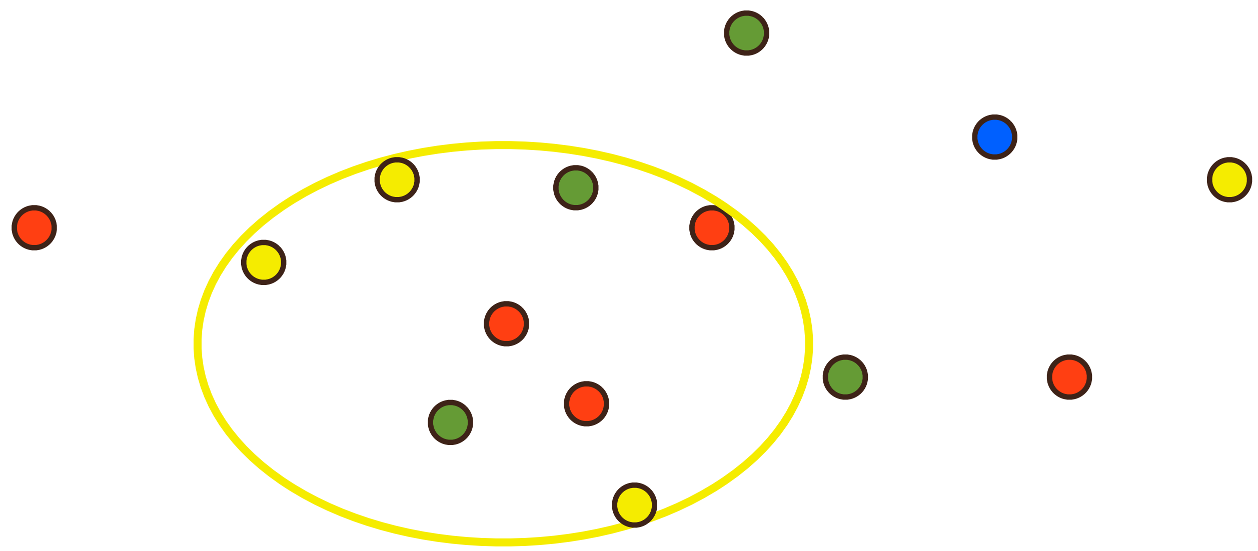
Alpha-conversion



Alpha-conversion



Scope extension



Mobility

Data moves from caller to called

```
int k=6;
k=fac(k)+3;
```

```
int fac(int i)
  {if (i<2) return 1;
  else return i*fac(i-1);
  }
```

Example: call by value

Mobility

Data moves from caller to called

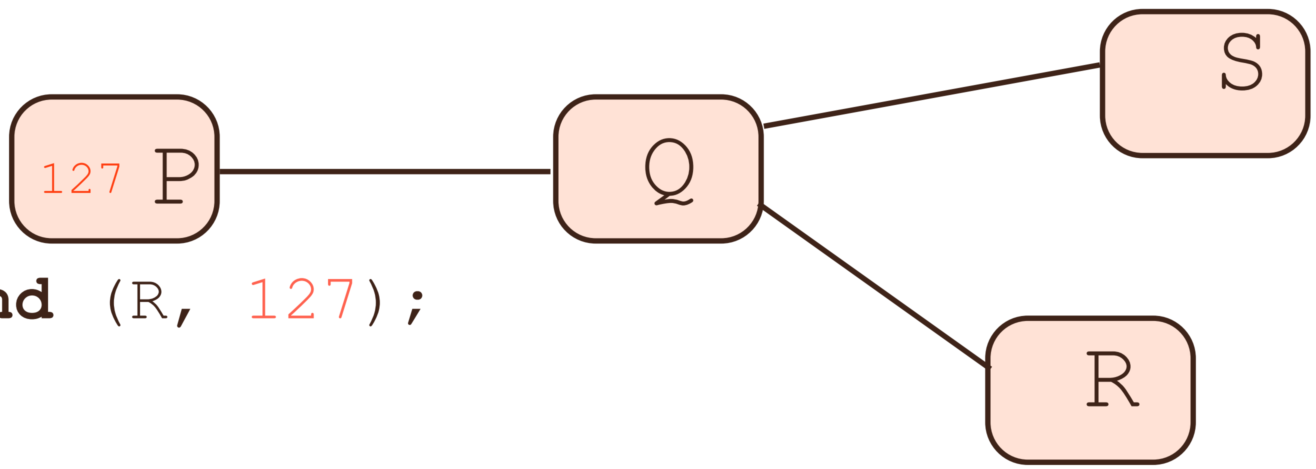
```
int k=6;
k=fac(k)+3;
```

```
int fac(int 6)
  {if (6<2) return 1;
  else return 6*fac(6-1);
  }
```

Example: call by value

Mobility

Data moves in a network

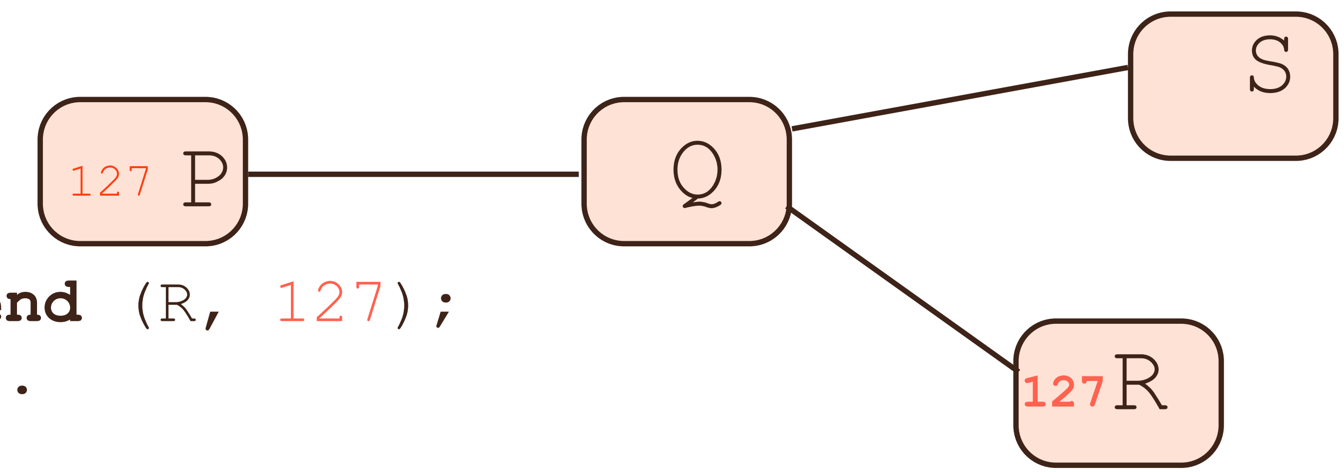



```
send (R, 127);
...
```

```
receive (P, x);
if (x>0) ...
```

Mobility

Data moves in a network




```
send (R, 127);
...
```

```
receive (P, 127);
if (127>0) ...
```

Mobility of Scopes

What happens when a scoped thing moves out of its boundary?

Example: call by reference

```
{int k=3;
  foo(k); if k==4 ...}
```

```
int foo(ref int i)
  {if (i<2) i++;}
```

Mobility of Scopes

What happens when a scoped thing moves out of its boundary?

Example: call by reference

```
{int k=3;
  foo(k); if k==4 ...}
```

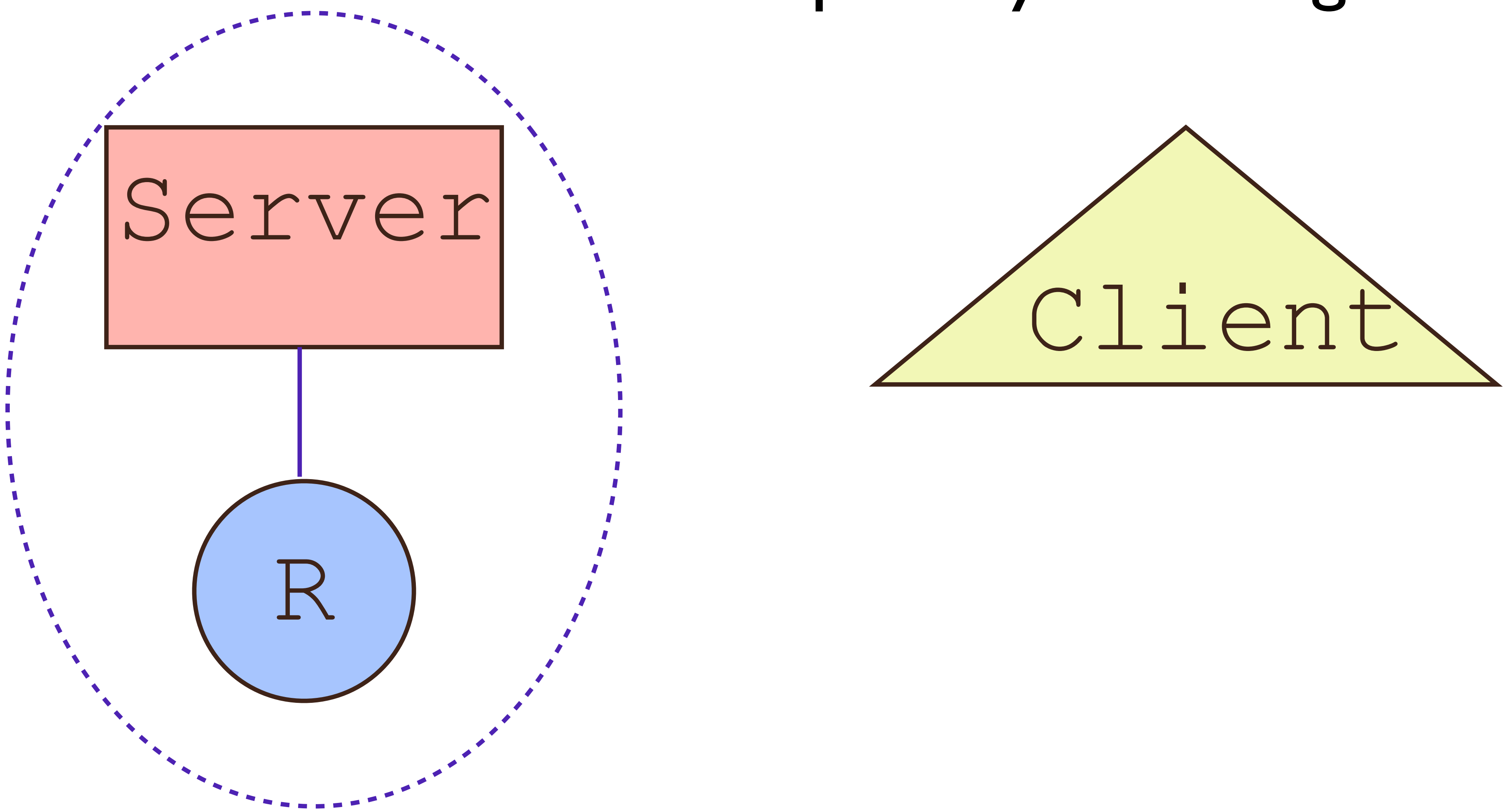
```
int foo(ref int k)
  {if (k<2) k++;}
```

Scope of `k` is increased to old `i`!

Mobility of Scopes

Example: transfer access to local resource

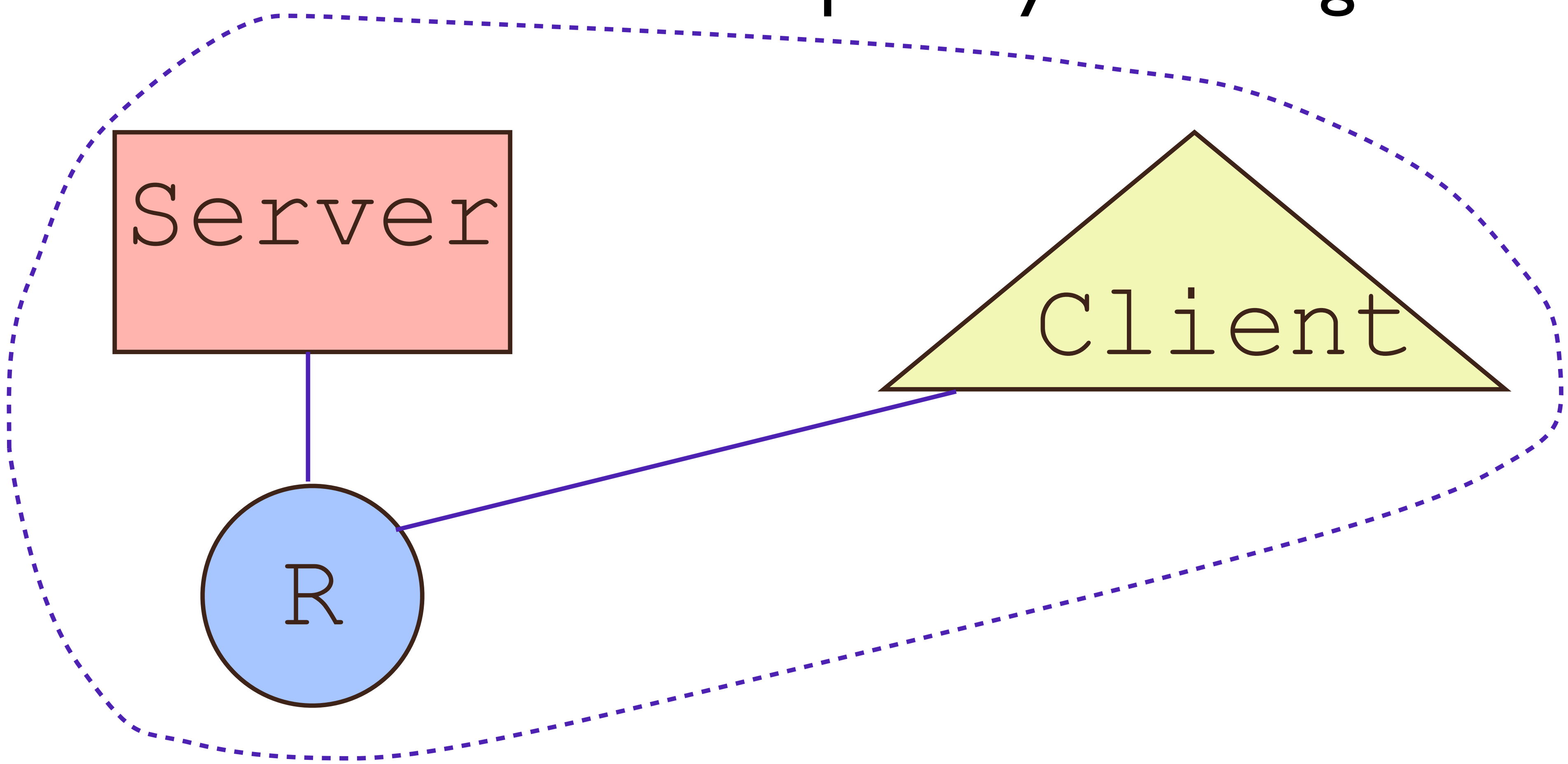
The access to `R` is in a scope only including `Server`



Mobility of Scopes

Example: transfer access to local resource

The access to `R` is in a scope only including `Server`

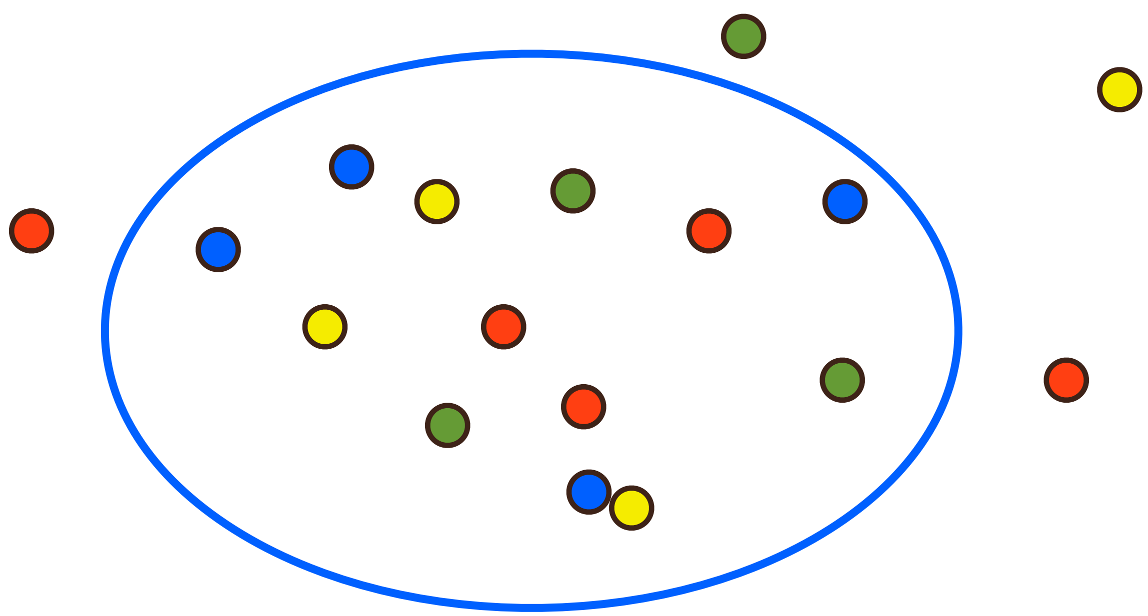


The scope includes also `Client`!

Mobility of Scopes

Universal law #3: Scope Extrusion

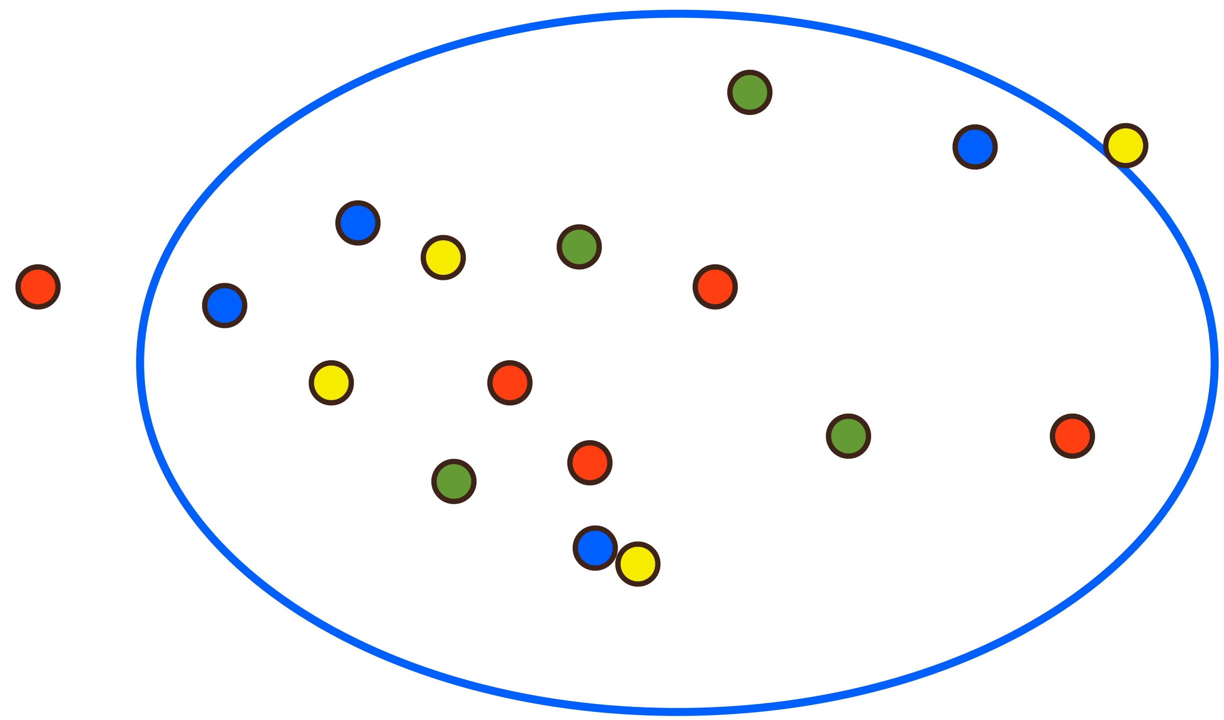
When a scoped item is moved,
the scope follows the item



Mobility of Scopes

Universal law #3: Scope Extrusion

When a scoped item is moved,
the scope follows the item



Mobility of Scopes

What happens when a thing moves into a scope?

```
foo(k); if k==4 ...
```

```
int foo(ref int i)
  {int k=0;
   if (i<k) i++;}
```

Mobility of Scopes

What happens when a thing moves into a scope?

```
foo(k); if k==4 ...
```

```
int foo(ref int k)
  {int k=0;
   if (k<k) k++;}
```

Mobility of Scopes

What happens when a thing moves into a scope?

```
foo(k); if k==4 ...
```

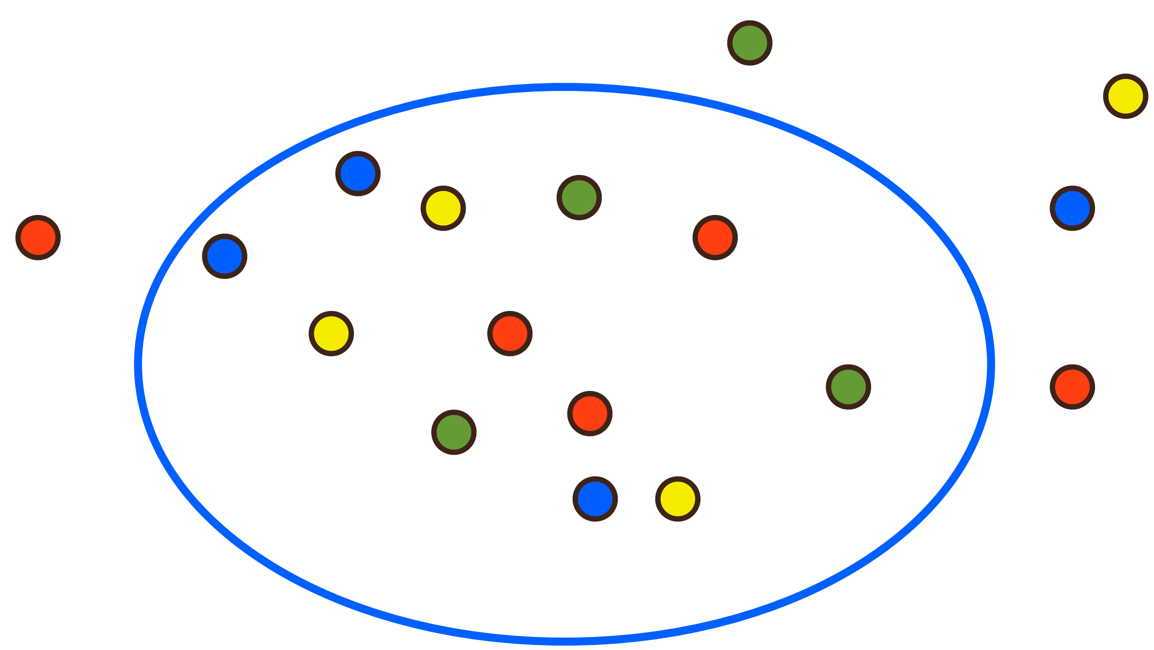
```
int foo(ref int k)
  {int m=0;
   if (k<m) k++;}
```

The scope is alpha-converted!

Mobility of Scopes

Universal law #4: Scope Intrusion

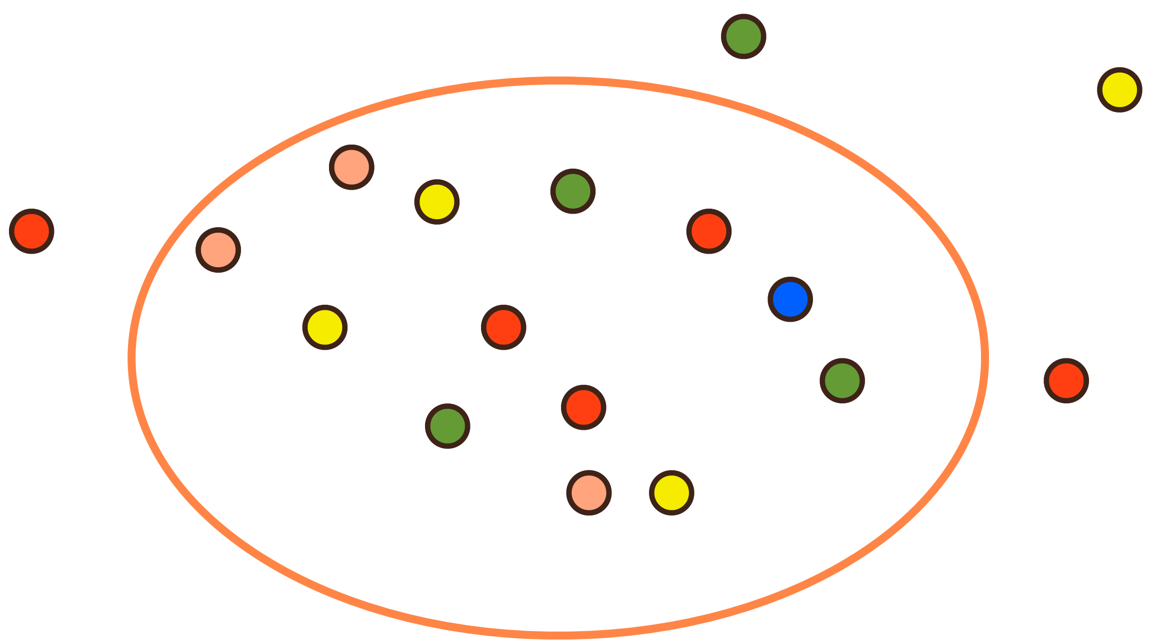
When an item is moved inside a scope, that scope is alpha-converted

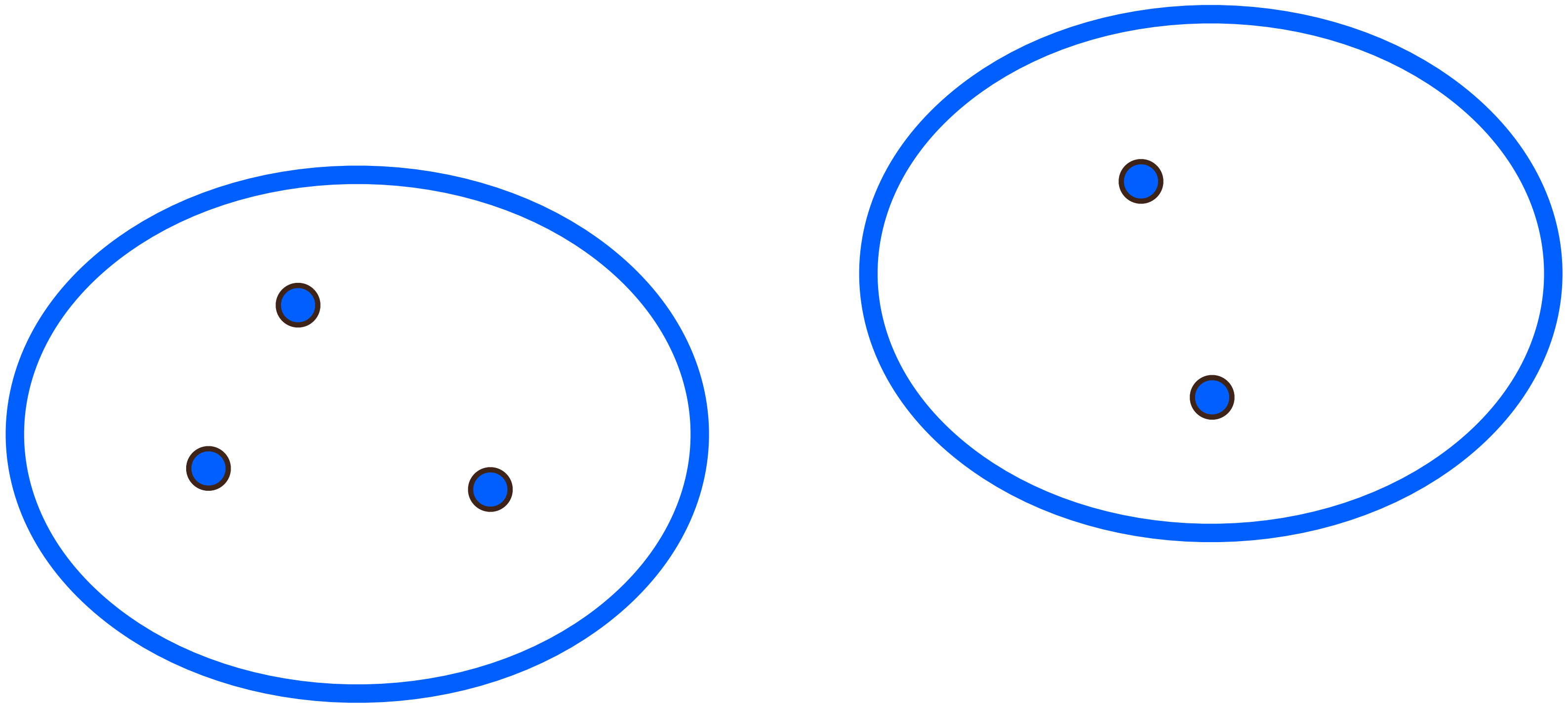


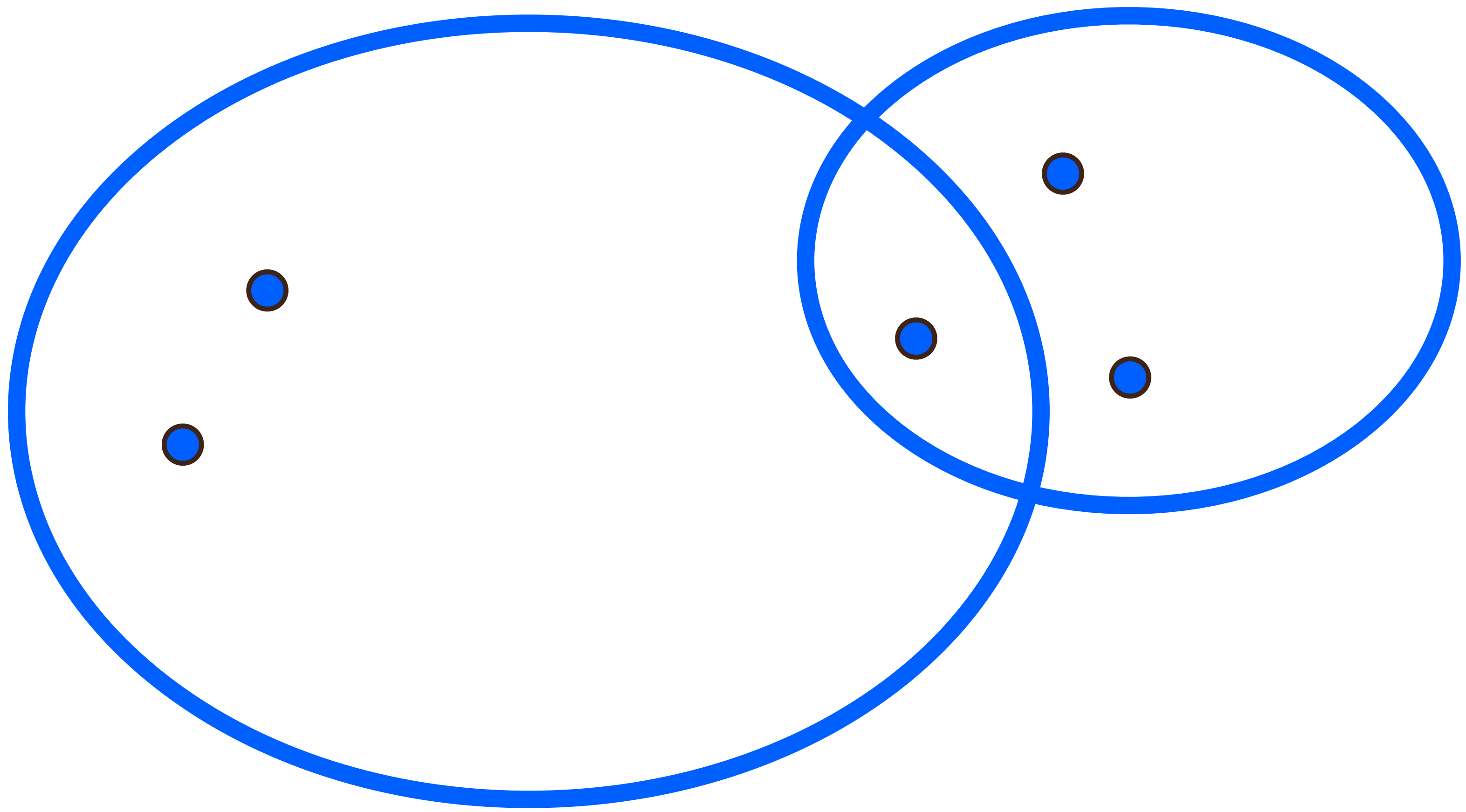
Mobility of Scopes

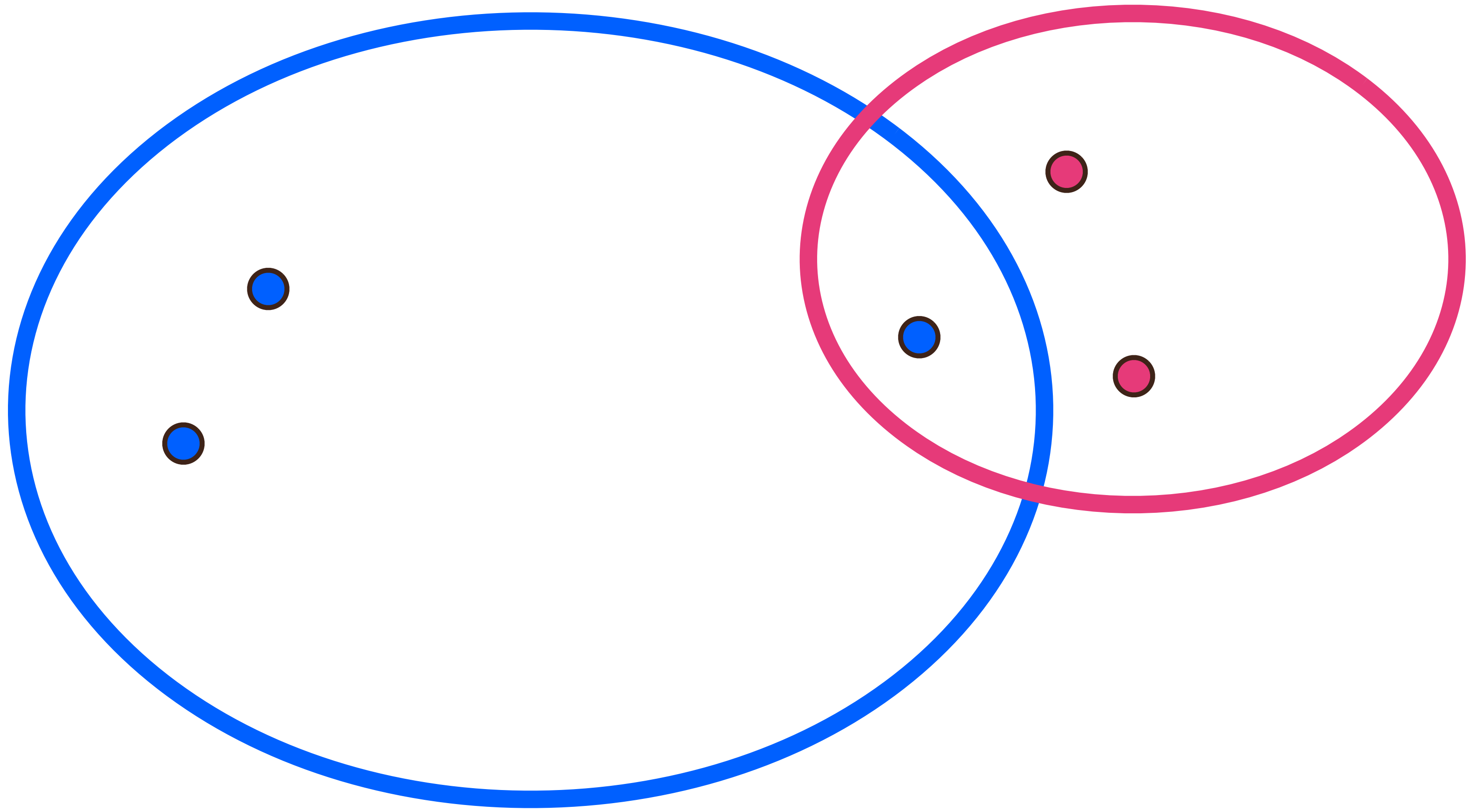
Universal law #4: Scope Intrusion

When an item is moved inside a scope, that scope is alpha-converted









A minimal model

Assume a set of names $a, b, \ldots z$

The agents $P, Q, \ldots$ are of the following forms:

$\overline{a}u.P$	Output, send u along a
$a(x).P$	Input for x along a
$P \mid Q$	P and Q in parallel
$(\nu z)P$	Restriction: z is local in P

Example

$\overline{a}z.P$ Output z along a

$a(x).\overline{b}x.Q$ Input something on a, then output it on b

Example

$\overline{a}z.P$ Output z along a

$a(x).\overline{b}x.Q$ Input something on a, then output it on b

$$\overline{a}z.P \mid a(x).\overline{b}x.Q$$

Example

$\overline{a}z.P$ Output z along a

$a(x).\overline{b}x.Q$ Input something on a, then output it on b

"Does not occur in"

$P \mid \quad \overline{b}z.Q$ Assuming $x\#Q$

$\overline{a}z.P \mid a(x).\overline{b}x.Q$

Example

$\overline{a}z.P$ — Output z along a

$a(x).\overline{b}x.Q$ — Input something on a, then output it on b

"Does not occur in"

$P \mid \quad \overline{b}z.Q$ — Assuming $x\#Q$

$$\overline{a}z.P \mid a(x).\overline{b}x.Q \longrightarrow P \mid \overline{b}z.Q$$

Rules for Transitions

$$P \longrightarrow Q$$

Means that P can **evolve** into Q through an action that is **internal** to P, possibly an interaction between components of P.

Rules for Transitions

$$P \longrightarrow Q$$

Means that P can **evolve** into Q through an action that is **internal** to P, possibly an interaction between components of P.

What about **compositionality**? How can we calculate the transitions of $T|U$ in terms of the transitions of T and U?

Bad News:

This is clearly impossible:

$\overline{a}u.P$ $\overline{b}u.P$

Have the same transitions
(namely none)

Bad News:

This is clearly impossible:

$$\overline{a}u.P \mid a(x).Q \qquad \overline{b}u.P \mid a(x).Q$$

$$\downarrow \qquad\qquad \not\downarrow$$

$$P \mid Q$$

Have different transitions

Labelled Transitions

Solution: Introduce **labels** on transitions to signify **output** and **input** actions.

$$P \xrightarrow{\overline{a}u} P'$$

Says P can **output** u along a and move to state P'

$$P \xrightarrow{au} P'$$

Says P can **input** u along a and move to state P'

Some Rules

$$\overline{a}u.P \xrightarrow{\overline{a}u} P$$

$$a(x).P \xrightarrow{au} P[x := u]$$

Substitution: Replace all x by u, while alpha-converting any scopes of u to take care of scope intrusion

Some Rules

$$\overline{a}u.P \xrightarrow{\overline{a}u} P$$

$$a(x).P \xrightarrow{au} P[x := u]$$

Substitution: Replace all x by u, while alpha-converting any scopes of u to take care of scope intrusion

Example:

$$a(x).\overline{b}x.Q \xrightarrow{au} \overline{b}u.Q$$

Assuming $x\#Q$

Communication rule

$$\frac{P \xrightarrow{\overline{a}u} P', \quad Q \xrightarrow{au} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Example:

$$\overline{a}z.P \mid a(x).\overline{b}x.Q \xrightarrow{\tau} P \mid \overline{b}z.Q$$

Communication rule

$$\frac{P \xrightarrow{\overline{a}u} P', \quad Q \xrightarrow{au} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Example:

$$\overline{a}\,z.P \mid a(x).\overline{b}x.Q \xrightarrow{\tau} P \mid \overline{b}z.Q$$

Now, what about scope extrusions?

Blackboard in Robin's office, April 1987

$$\overline{a}x.P \xrightarrow{\overline{a}x} P \qquad a(y).Q \xrightarrow{a(y)} Q$$

$$\overline{a}x.P \mid a(y).Q \xrightarrow{\tau} P \mid Q\{x/y\}$$

Blackboard in Robin's office, April 1987

Value Passing

$$\overline{a}x.P \xrightarrow{\overline{a}x} P \qquad a(y).Q \xrightarrow{a(y)} Q$$

$$\overline{a}x.P \mid a(y).Q \xrightarrow{\tau} P \mid Q\{x/y\}$$

Blackboard in Robin's office, April 1987

$$\overline{a}x.P \xrightarrow{\overline{a}x} P \qquad a(y).Q \xrightarrow{a(y)} Q$$

$$\overline{a}x.P \mid a(y).Q \xrightarrow{\tau} P \mid Q\{x/y\}$$

$$(\overline{a}x.P)\backslash x$$

Blackboard in Robin's office, April 1987

$$\overline{a}x.P \xrightarrow{\overline{a}x} P \qquad a(y).Q \xrightarrow{a(y)} Q$$

$$\overline{a}x.P \mid a(y).Q \xrightarrow{\tau} P \mid Q\{x/y\}$$

$$(\overline{a}x.P)\backslash x \xrightarrow{\overline{a}(x)} P$$

Blackboard in Robin's office, April 1987

$$\overline{a}x.P \xrightarrow{\overline{a}x} P \qquad a(y).Q \xrightarrow{a(y)} Q$$

$$\overline{a}x.P \mid a(y).Q \xrightarrow{\tau} P \mid Q\{x/y\}$$

$$(\overline{a}x.P)\backslash x \xrightarrow{\overline{a}(x)} P$$

$$(\overline{a}x.P)\backslash x \mid a(y).Q$$

Blackboard in Robin's office, April 1987

$$\overline{a}x.P \xrightarrow{\overline{a}x} P \qquad a(y).Q \xrightarrow{a(y)} Q$$

$$\overline{a}x.P \mid a(y).Q \xrightarrow{\tau} P \mid Q\{x/y\}$$

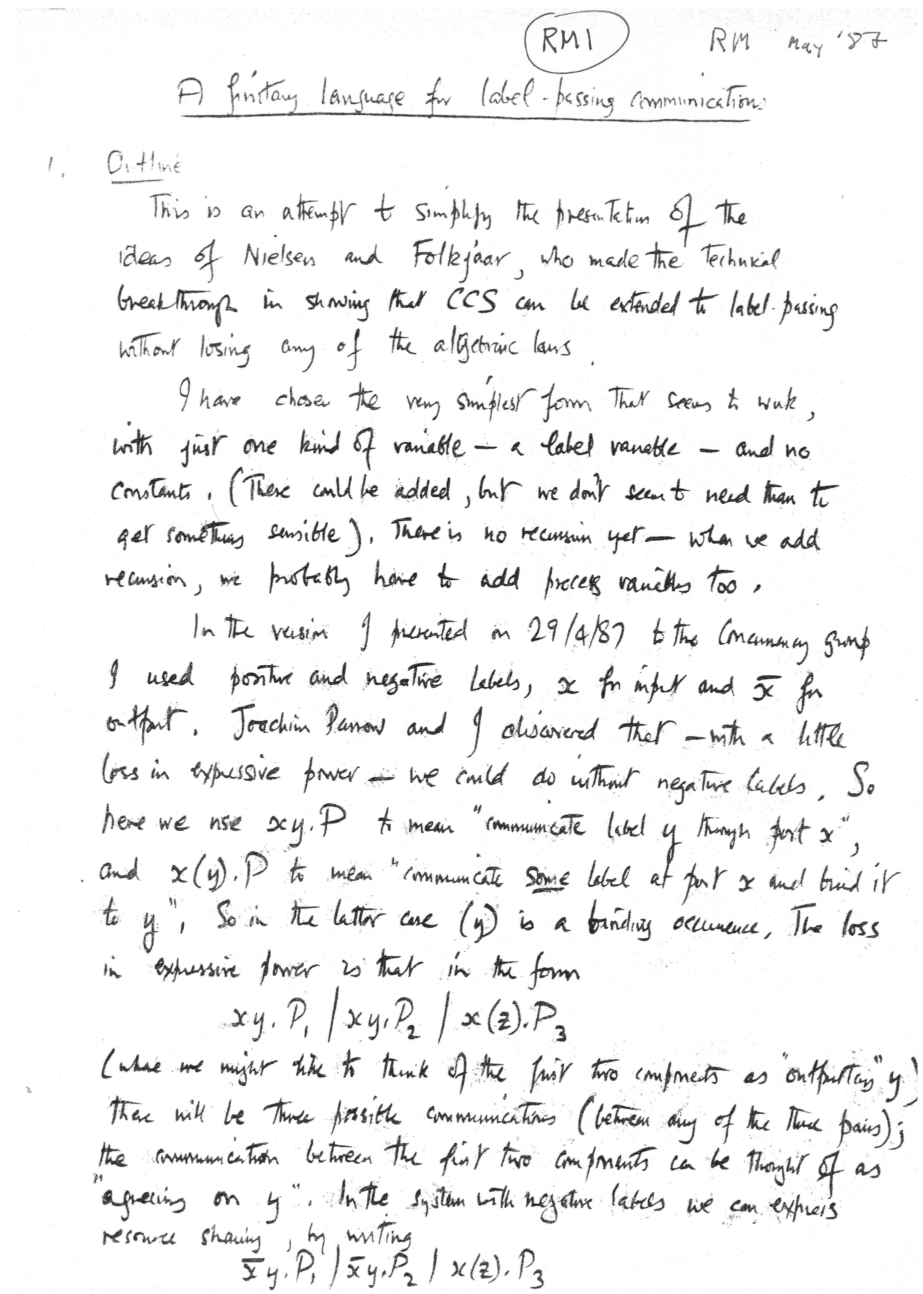
$$(\overline{a}x.P)\backslash x \xrightarrow{\overline{a}(x)} P$$

$$(\overline{a}x.P)\backslash x \mid a(y).Q \xrightarrow{\tau} (P \mid Q\{x/y\})\backslash x$$

The very first written note by Robin on what was to become the pi-calculus.

What do you think Robin did in the very first sentence?

1) Explained the main idea
x) Explained the motivation
2) Gave most of the credit to someone else



RM1 RM May '87

A finitary language for label-passing communication

1. Outline

This is an attempt to simplify the presentation of the ideas of Nielsen and Folkjaar, who made the technical breakthrough in showing that CCS can be extended to label-passing without losing any of the algebraic laws.

I have chosen the very simplest form that seems to work, with just one kind of variable — a label variable — and no constants. (These could be added, but we don't seem to need them to get something sensible). There is no recursion yet — when we add recursion, we probably have to add process variables too.

In the version I presented on 29/4/87 to the Concurrency group I used positive and negative labels, x for input and $\bar{x}$ for output. Joachim Parrow and I observed that — with a little loss in expressive power — we could do without negative labels. So here we use $xy.P$ to mean "communicate label y through port x", and $x(y).P$ to mean "communicate some label at port x and bind it to y". So in the latter case (y) is a binding occurrence. The loss in expressive power is that in the form

$$xy.P_1 \mid xy.P_2 \mid x(z).P_3$$

(where we might like to think of the first two components as "outputting" y) there will be three possible communications (between any of the three pairs); the communication between the first two components can be thought of as "agreeing on y". In the system with negative labels we can express resource sharing, by writing

$$\bar{x}y.P_1 \mid \bar{x}y.P_2 \mid x(z).P_3$$

RM1 RM May '87

A primitary language for label-passing communication

1. Outline

This is an attempt to simplify the presentation of the ideas of Nielsen and Folkjaar, who made the technical breakthrough in showing that CCS can be extended to label-passing without losing any of the algebraic laws.

I have chosen the very simplest form that seems to work, with just one kind of variable — a label variable — and no constants. (These could be added, but we don't seem to need them to get something sensible). There is no recursion yet — when we add recursion, we probably have to add process variables too.

In the version I presented on 29/4/87 to the Concurrency group I used positive and negative labels, x for input and $\bar{x}$ for output. Joachim Parrow and I observed that — with a little

(RM1) RM May '87

A finitary language for label-passing communication

1. Outline

This is an attempt to simplify the presentation of the ideas of Nielsen and Folkjaar, who made the technical breakthrough in showing that CCS can be extended to label-passing without losing any of the algebraic laws.

I have chosen the very simplest form that seems to work,

> "This is an attempt to simplify the presentation of the ideas of Nielsen and Folkjaar [sic], who made the technical breakthrough in showing that CCS can be extended to label-passing without losing any of the algebraic laws"

In the version I presented on 29/4/87 to the Concurrency group I used positive and negative labels, x for input and $\bar{x}$ for output. Joachim Parrow and I observed that – with a little

Example

$(\nu z)\, \overline{a}z.P$

Output a local z along a

$a(x).\overline{b}x.Q$

Input something on a,
then output it on b

Example

$(\nu z)\,\overline{a}z.P$ Output a local z along a

$a(x).\overline{b}x.Q$ Input something on a, then output it on b

$$(\nu z)\,\overline{a}\,z.\,P \mid a(x).\,\overline{b}x\,.\,Q$$

Example

$(\nu z)\,\overline{a}z.P$ Output a local z along a

$a(x).\overline{b}x.Q$ Input something on a, then output it on b

(νz)

$P \mid \qquad \overline{b}z.Q$

$(\nu z)\overline{a}z.P \mid a(x).\overline{b}x.Q$

Example

$(\nu z)\,\overline{a}z.P$ Output a local z along a

$a(x).\overline{b}x.Q$ Input something on a, then output it on b

Because of Scope Extrusion!

$$(\nu z)(P \mid \overline{b}z.Q)$$

$$(\nu z)\overline{a}z.P \mid a(x).\overline{b}x.Q$$

Example

$$(\nu z)\,\overline{a}z.P$$

Output a local z along a

$$a(x).\overline{b}x.Q$$

Input something on a, then output it on b

Because of Scope Extrusion!

$$(\nu z)(P| \qquad \overline{b}z.Q\ )$$

$$(\nu z)\overline{a}z.P|\ a(x).\overline{b}x.Q \xrightarrow{\tau} (\nu z)(P|\overline{b}z.Q)$$

Scope Extrusion Rules

Scope Opening

$$\frac{P \xrightarrow{\overline{a}z} P'}{(\nu z)P \xrightarrow{\overline{a}(\nu z)z} P'} \; a \neq z$$

Scope Closing

$$\frac{P \xrightarrow{\overline{a}(\nu z)z} P', \quad Q \xrightarrow{az} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')} \; z \# Q$$

Scope Extrusion Rules

Scope Opening

$$\frac{P \xrightarrow{\overline{a}z} P'}{(\nu z)P \xrightarrow{\overline{a}(\nu z)z} P'} \; a \neq z$$

A new kind of action signifying an output of a scoped z along a

Scope Closing

$$\frac{P \xrightarrow{\overline{a}(\nu z)z} P', \quad Q \xrightarrow{az} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')} \; z \# Q$$

Scope Extrusion Rules

Scope Opening

$$\frac{P \xrightarrow{\overline{a}z} P'}{(\nu z)P \xrightarrow{\overline{a}(\nu z)z} P'}\; a \neq z$$

A new kind of action signifying an output of a scoped z along a

Note that the scope has disappeared from the agent. Instead it sits on the transition label.

Scope Closing

$$\frac{P \xrightarrow{\overline{a}(\nu z)z} P', \quad Q \xrightarrow{az} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}\; z\#Q$$

Scope Extrusion Rules

Scope Opening

$$\frac{P \xrightarrow{\overline{a}z} P'}{(\nu z)P \xrightarrow{\overline{a}(\nu z)z} P'} \; a \neq z$$

Scope Closing

$$\frac{P \xrightarrow{\overline{a}(\nu z)z} P', \quad Q \xrightarrow{az} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')} \; z\#Q$$

The scope reappears in the agent.

Scope Extrusion Example

$$(\nu z)\overline{a}z.P \mid a(x).\overline{b}x.Q \to (\nu z)(P|\overline{b}z.Q)$$

Scope Extrusion Example

$$\overline{a}z.P \xrightarrow{\overline{a}z} P$$

$$\overline{a}u.P \xrightarrow{\overline{a}u} P'$$

$$(\nu z)\overline{a}z.P \mid a(x).\overline{b}x.Q \rightarrow (\nu z)(P \mid \overline{b}z.Q)$$

Scope Extrusion Example

$$\overline{a}z.P \xrightarrow{\overline{a}z} P$$

$$(\nu z)\overline{a}z.P \xrightarrow{\overline{a}(\nu z)z} P$$

$$\frac{P \xrightarrow{\overline{a}z} P'}{(\nu z)P \xrightarrow{\overline{a}(\nu z)z} P'}$$

$$(\nu z)\overline{a}z.P \mid a(x).\overline{b}x.Q \rightarrow (\nu z)(P \mid \overline{b}z.Q)$$

Scope Extrusion Example

$$\overline{a}z.P \xrightarrow{\overline{a}z} P$$

$$(\nu z)\overline{a}z.P \xrightarrow{\overline{a}(\nu z)z} P$$

$$a(x).\overline{b}x.Q \xrightarrow{az} bz.Q$$

$$a(x).P \xrightarrow{au} P'[x := u]$$

$$(\nu z)\overline{a}z.P \,|\, a(x).\overline{b}x.Q \longrightarrow (\nu z)(P|\overline{b}z.Q)$$

Scope Extrusion Example

$$\overline{a}z.P \stackrel{\overline{a}z}{\rightarrow} P$$

$$(\nu z)\overline{a}z.P \stackrel{\overline{a}(\nu z)z}{\rightarrow} P$$

$$a(x).\overline{b}x.Q \stackrel{az}{\rightarrow} bz.Q$$

$$\frac{P \stackrel{\overline{a}(\nu z)z}{\longrightarrow} P' \qquad Q \stackrel{az}{\longrightarrow} Q'}{P|Q \longrightarrow (\nu z)(P'|Q')}$$

$$(\nu z)\overline{a}z.P \,|\, a(x).\overline{b}x.Q \longrightarrow (\nu z)(P|\overline{b}z.Q)$$

All the rules

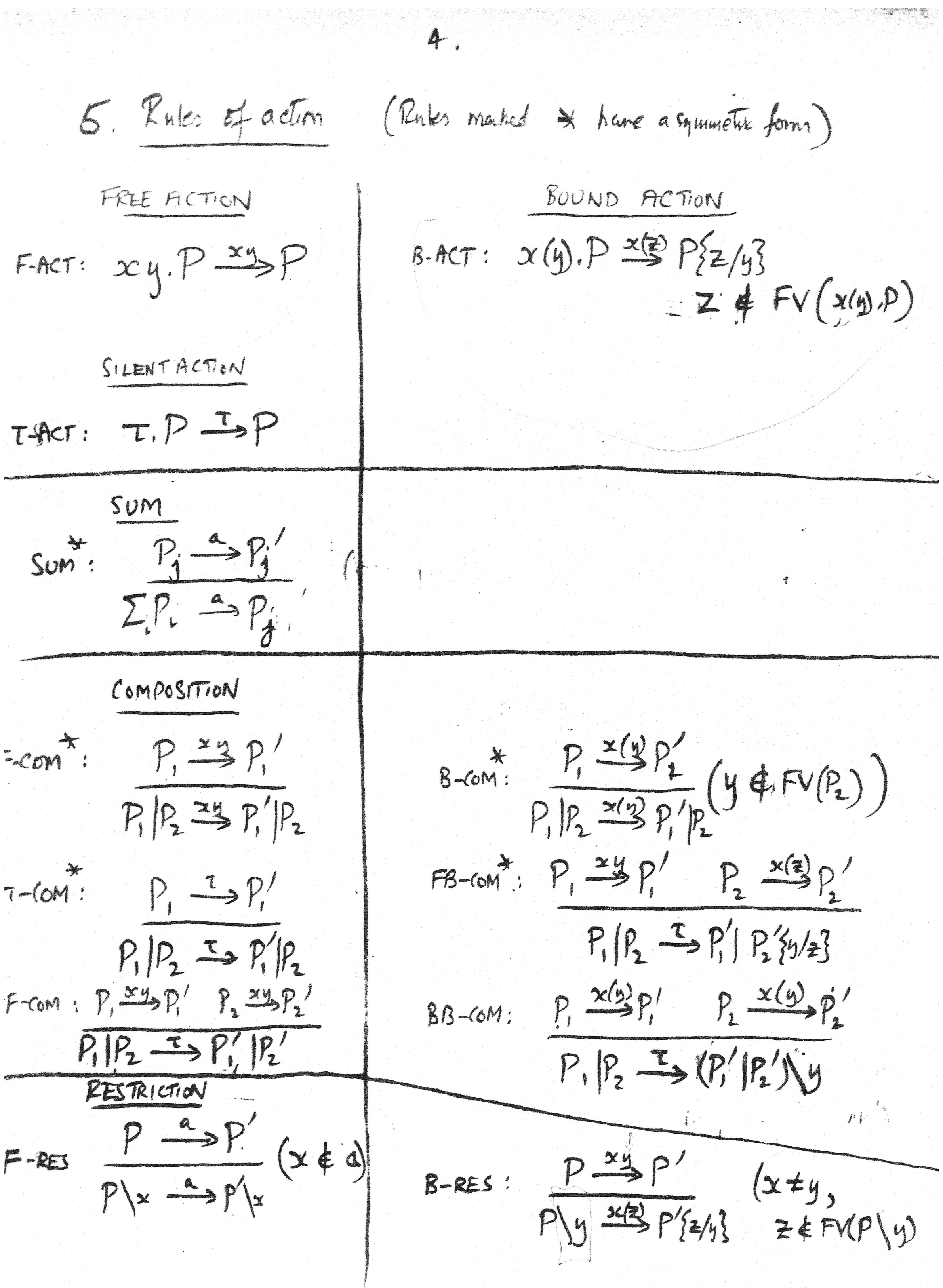
$$\overline{a}u\,.\,P \xrightarrow{\overline{a}u} P \qquad\qquad a(x)\,.\,P \xrightarrow{au} P[x := u]$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathrm{bn}(\alpha)\#Q \qquad\qquad \frac{P \xrightarrow{\overline{a}u} P', \quad Q \xrightarrow{au} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\frac{P \xrightarrow{\overline{a}(\nu z)z} P', \quad Q \xrightarrow{az} Q'}{P \mid Q \xrightarrow{\tau} (\nu z)(P' \mid Q')}\ z\#Q$$

$$\frac{P \xrightarrow{\alpha} P'}{(\nu u)P \xrightarrow{\alpha} (\nu u)P'}\ u\#\alpha \qquad\qquad \frac{P \xrightarrow{\overline{a}z} P'}{(\nu z)P \xrightarrow{\overline{a}(\nu z)z} P'}\ a \neq z$$

The first pi-calculus semantics (May '87)!



4.

5. Rules of action (Rules marked * have a symmetric form)

FREE ACTION

F-ACT: $\bar{x}y.P \xrightarrow{\bar{x}y} P$

SILENT ACTION

τ-ACT: $\tau.P \xrightarrow{\tau} P$

BOUND ACTION

B-ACT: $x(y).P \xrightarrow{x(z)} P\{z/y\}$ $\quad z \notin FV(x(y).P)$

SUM

SUM*: $\dfrac{P_j \xrightarrow{a} P_j'}{\sum_i P_i \xrightarrow{a} P_j'}$

COMPOSITION

F-COM*: $\dfrac{P_1 \xrightarrow{\bar{x}y} P_1'}{P_1|P_2 \xrightarrow{\bar{x}y} P_1'|P_2}$

τ-COM*: $\dfrac{P_1 \xrightarrow{\tau} P_1'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2}$

F-COM: $\dfrac{P_1 \xrightarrow{\bar{x}y} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'}$

B-COM*: $\dfrac{P_1 \xrightarrow{x(y)} P_1'}{P_1|P_2 \xrightarrow{x(y)} P_1'|P_2}$ $\quad (y \notin FV(P_2))$

FB-COM*: $\dfrac{P_1 \xrightarrow{\bar{x}y} P_1' \quad P_2 \xrightarrow{x(z)} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'\{y/z\}}$

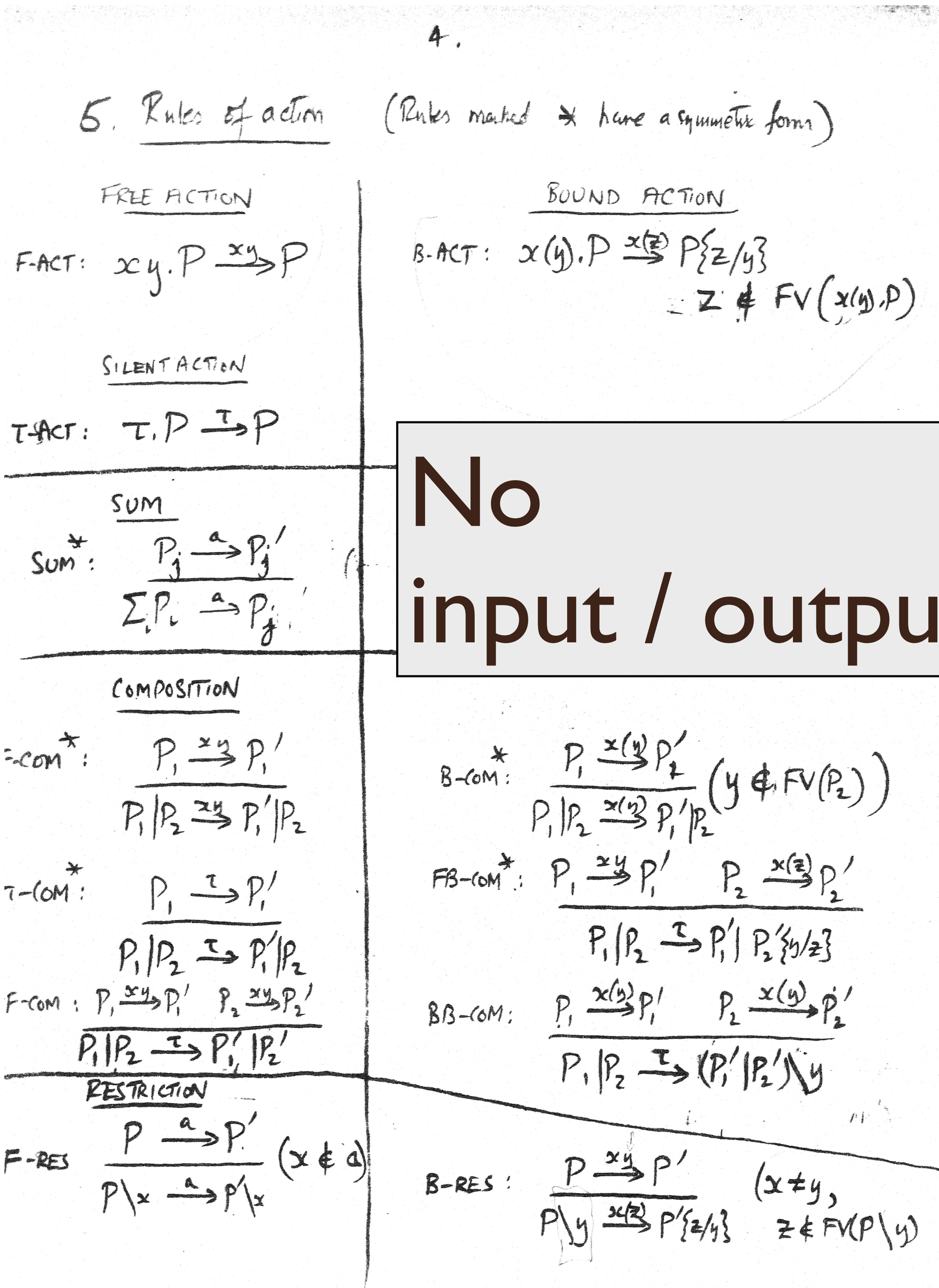
BB-COM: $\dfrac{P_1 \xrightarrow{\bar{x}(y)} P_1' \quad P_2 \xrightarrow{x(y)} P_2'}{P_1|P_2 \xrightarrow{\tau} (P_1'|P_2')\backslash y}$

RESTRICTION

F-RES: $\dfrac{P \xrightarrow{a} P'}{P\backslash x \xrightarrow{a} P'\backslash x}$ $\quad (x \notin a)$

B-RES: $\dfrac{P \xrightarrow{\bar{x}y} P'}{P\backslash y \xrightarrow{\bar{x}(z)} P'\{z/y\}}$ $\quad (x \neq y,\ z \notin FV(P\backslash y))$

The first pi-calculus semantics (May '87)!



4.

5. Rules of action (Rules marked * have asymmetric form)

FREE ACTION

F-ACT: $xy.P \xrightarrow{xy} P$

SILENT ACTION

T-ACT: $\tau.P \xrightarrow{\tau} P$

SUM

SUM*: $\dfrac{P_j \xrightarrow{a} P_j'}{\Sigma_i P_i \xrightarrow{a} P_j'}$

COMPOSITION

F-COM*: $\dfrac{P_1 \xrightarrow{xy} P_1'}{P_1 | P_2 \xrightarrow{xy} P_1' | P_2}$

T-COM*: $\dfrac{P_1 \xrightarrow{\tau} P_1'}{P_1 | P_2 \xrightarrow{\tau} P_1' | P_2}$

F-COM: $\dfrac{P_1 \xrightarrow{xy} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1 | P_2 \xrightarrow{\tau} P_1' | P_2'}$

RESTRICTION

F-RES: $\dfrac{P \xrightarrow{a} P'}{P\backslash x \xrightarrow{a} P'\backslash x}$ $(x \notin a)$

BOUND ACTION

B-ACT: $x(y).P \xrightarrow{x(z)} P\{z/y\}$ $z \notin FV(x(y).P)$

B-COM*: $\dfrac{P_1 \xrightarrow{x(y)} P_1'}{P_1 | P_2 \xrightarrow{x(y)} P_1' | P_2}$ $(y \notin FV(P_2))$

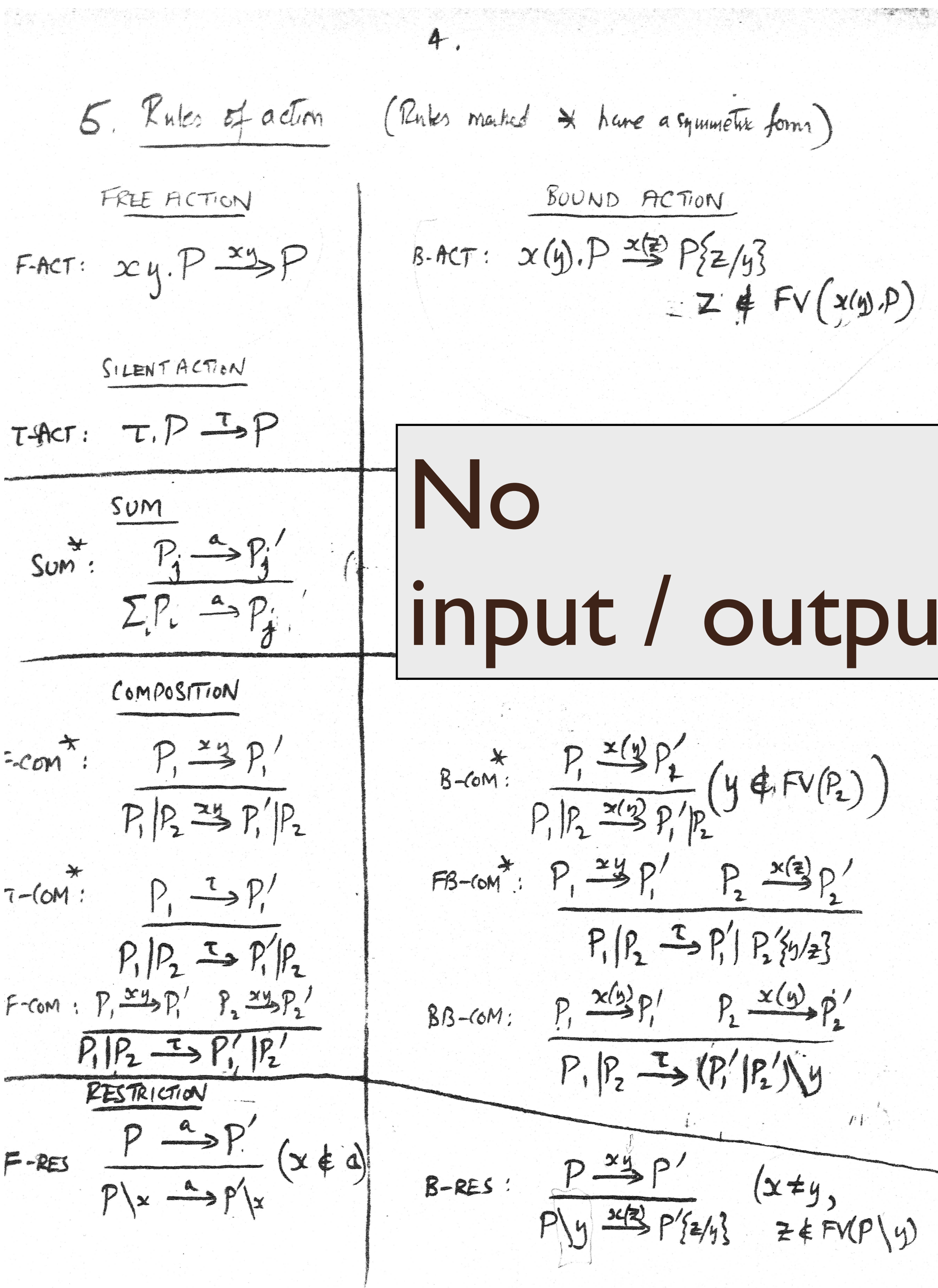
FB-COM*: $\dfrac{P_1 \xrightarrow{xy} P_1' \quad P_2 \xrightarrow{x(z)} P_2'}{P_1 | P_2 \xrightarrow{\tau} P_1' | P_2'\{y/z\}}$

BB-COM: $\dfrac{P_1 \xrightarrow{x(y)} P_1' \quad P_2 \xrightarrow{x(y)} P_2'}{P_1 | P_2 \xrightarrow{\tau} (P_1' | P_2')\backslash y}$

B-RES: $\dfrac{P \xrightarrow{xy} P'}{P\backslash y \xrightarrow{x(z)} P'\{z/y\}}$ $(x \neq y,\ z \notin FV(P\backslash y))$

No
input / output

The first pi-calculus semantics (May '87)!



4.

5. Rules of action (Rules marked * have asymmetric form)

FREE ACTION

F-ACT: $xy.P \xrightarrow{xy} P$

BOUND ACTION

B-ACT: $x(y).P \xrightarrow{x(z)} P\{z/y\}$, $z \notin FV(x(y).P)$

SILENT ACTION

T-ACT: $\tau.P \xrightarrow{\tau} P$

SUM

SUM*: $\dfrac{P_j \xrightarrow{a} P_j'}{\Sigma_i P_i \xrightarrow{a} P_j'}$

COMPOSITION

F-COM*: $\dfrac{P_1 \xrightarrow{xy} P_1'}{P_1|P_2 \xrightarrow{xy} P_1'|P_2}$

B-COM*: $\dfrac{P_1 \xrightarrow{x(y)} P_1'}{P_1|P_2 \xrightarrow{x(y)} P_1'|P_2}$ $(y \notin FV(P_2))$

T-COM*: $\dfrac{P_1 \xrightarrow{\tau} P_1'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2}$

FB-COM*: $\dfrac{P_1 \xrightarrow{xy} P_1' \quad P_2 \xrightarrow{x(z)} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'\{y/z\}}$

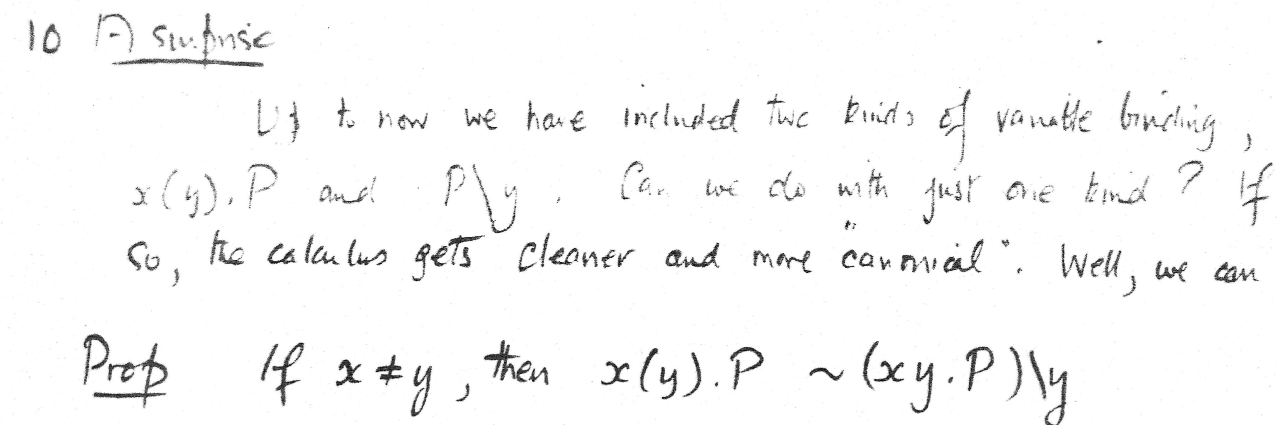
F-COM: $\dfrac{P_1 \xrightarrow{xy} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'}$

BB-COM: $\dfrac{P_1 \xrightarrow{x(y)} P_1' \quad P_2 \xrightarrow{x(y)} P_2'}{P_1|P_2 \xrightarrow{\tau} (P_1'|P_2')\backslash y}$

RESTRICTION

F-RES: $\dfrac{P \xrightarrow{a} P'}{P\backslash x \xrightarrow{a} P'\backslash x}$ $(x \notin a)$

B-RES: $\dfrac{P \xrightarrow{xy} P'}{P\backslash y \xrightarrow{x(z)} P'\{z/y\}}$ $(x \neq y,\ z \notin FV(P\backslash y))$



10 A Surprise

Up to now we have included two kinds of variable binding, x(y).P and P\y. Can we do with just one kind? If so, the calculus gets cleaner and more "canonical". Well, we can!

Prop If x ≠ y, then x(y).P ~ (xy.P)\y

No input / output

A Surprise

Up to now we have included the two kinds of variable binding, $x(y).P$ and $P\backslash y$. Can we do with just one kind? If so, the calculus gets cleaner and more "canonical". Well, we can!

Prop If $x \neq y$, then $x(y).P \sim (xy.P)\backslash y$

The first pi-calculus semantics (May '87)!

4.

5. Rules of action (Rules marked * have a symmetric form)

FREE ACTION | BOUND ACTION

T-COM: $P_1 \longrightarrow P_1'$ / $P_1|P_2 \xrightarrow{\tau} P_1'|P_2$

F-COM: $P_1 \xrightarrow{\bar{x}y} P_1' \quad P_2 \xrightarrow{xy} P_2'$ / $P_1|P_2 \xrightarrow{\tau} P_1'|P_2'$

$P_1|P_2 \xrightarrow{\tau} P_1'|P_2'\{y/z\}$

BB-COM: $P_1 \xrightarrow{\bar{x}(y)} P_1' \quad P_2 \xrightarrow{x(y)} P_2'$ / $P_1|P_2 \xrightarrow{\tau} (P_1'|P_2')\backslash y$

RESTRICTION

F-RES: $P \xrightarrow{a} P'$ / $P\backslash x \xrightarrow{a} P'\backslash x$ $(x \notin a)$

B-RES: $P \xrightarrow{\bar{x}y} P'$ / $P\backslash y \xrightarrow{\bar{x}(z)} P'\{z/y\}$ $(x \neq y,\ z \notin FV(P\backslash y))$

10 A surprise

Up to now we have included two kinds of variable binding, $x(y).P$ and $P\backslash y$. Can we do with just one kind? If so, the calculus gets cleaner and more "canonical". Well, we can!

...nds ...we ...gets ...n!

To be continued — we need to prove ~ a congruence, then associativity of | etc.

Prop If $x \neq y$, then $x(y).P \sim (xy.P)\backslash y$

Turned out to have:

- Wrong basic constructors
- Wrong definition of bisimulation
- No sensible algebraic laws

Turned out to have:

- Wrong basic constructors
- Wrong definition of bisimulation
- No sensible algebraic laws

Revision

Turned out to have:

- Wrong basic constructors
- Wrong definition of bisimulation
- No sensible algebraic laws

Revision

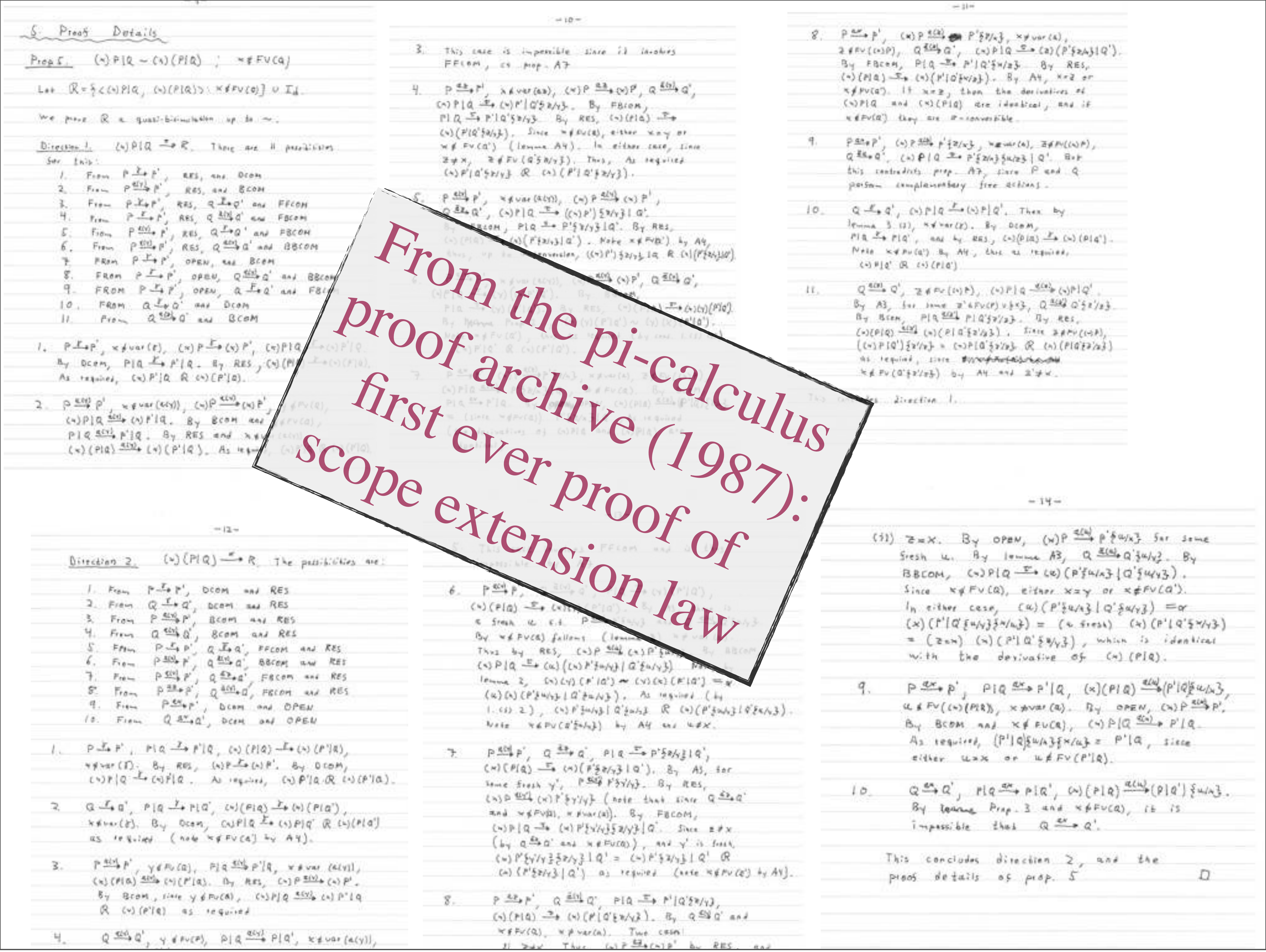
Establish properties

Turned out to have:

- Wrong basic constructors
- Wrong definition of bisimulation
- No sensible algebraic laws

Revision

Establish properties



From the pi-calculus proof archive (1987): first ever proof of scope extension law

Two years later...

Date: 12 Apr 89 15:13:18 BST
From: RM@ED.ECSVAX (Robin Milner)
Subject: How about this for a title and abstract?
To: jgp@ed.LFCS (N%"jgp@lfcs")
Message-Id: <"12-APR-1989 15:13:18">
Status: RO

Mobile processes (or the pi-calculus)

Robin Milner, Joachim Parrow, David Walker

Process calculi such as TCSP, ACP, CCS have not, on the whole, allowed for shifting contiguity among agents (though they allow them to bifurcate and to die). The purpose of this talk is to present a very basic calculus in which shifting contiguity, modelled by the use of names to communicate

Robin’s reply to my question ”why pi”?

I thought "process", or "pointer", or "parallel", but I also thought it a usable name -- if not too arrogant, and signifying that it aspires to primitivity like the lambda-calculus. You could also think of it as a near successor to the lambda calculus. Consider:

Robin’s reply to my question ”why pi”?

I thought "process", or "pointer", or "parallel", but I also thought it a usable name -- if not too arrogant, and signifying that it aspires to primitivity like the lambda-calculus. You could also think of it as a near successor to the lambda calculus. Consider:

mu-calculus ... this significantly exists

Robin’s reply to my question ”why pi”?

I thought "process", or "pointer", or "parallel", but I also thought it a usable name -- if not too arrogant, and signifying that it aspires to primitivity like the lambda-calculus. You could also think of it as a near successor to the lambda calculus. Consider:

mu-calculus ... this significantly exists

nu-calculus ... I thought we might have used this name, (nu standing for "name"), but mu and nu sound so alike.

Robin's reply to my question "why pi"?

I thought "process", or "pointer", or "parallel", but I also thought it a usable name -- if not too arrogant, and signifying that it aspires to primitivity like the lambda-calculus. You could also think of it as a near successor to the lambda calculus. Consider:

mu-calculus ... this significantly exists

nu-calculus ... I thought we might have used this name, (nu standing for "name"), but mu and nu sound so alike.

omicron calculus ... who would want that?

Robin’s reply to my question ”why pi”?

I thought "process", or "pointer", or "parallel", but I also thought it a usable name -- if not too arrogant, and signifying that it aspires to primitivity like the lambda-calculus. You could also think of it as a near successor to the lambda calculus. Consider:

mu-calculus ... this significantly exists

nu-calculus ... I thought we might have used this name,
(nu standing for "name"), but mu and nu
sound so alike.

omicron calculus ... who would want that?

which leads to

PI-CALCULUS

... I put it in parentheses to try it out ..

Exercise

- R represents a resource that can be started by communicating along a trigger port e.
- S represents a server controlling (ie handing out access to) the resource.
- C represents a client requesting the resource. There may be many clients.
- How ensure that the only clients who can start R are those who have been granted access by S?

Exercise

- R represents a resource that can be started by communicating along a trigger port e.
- S represents a server controlling (ie handing out access to) the resource.
- C represents a client requesting the resource. There may be many clients.
- How ensure that the only clients who can start R are those who have been granted access by S?

$$R = e \,.\, R'$$
$$S = (\nu e)(\overline{a}e \,.\, \ldots \mid R)$$
$$C = a(t) \ldots$$

Exercise

- F represent an agent enacting a function. It receives some value along a certain link and produces (a link to) some result:

$$F = f(x). \dots . \overline{r}v.\mathbf{0}$$

- A caller C calls F and waits for the result. There may be several callers.

$$C = \overline{f}u \,.\, r(z). \dots$$

- How make sure that that only the C who called F will receive the result of its call?

Exercise

- F represent an agent enacting a function. It receives some value along a certain link and produces (a link to) some result:

$$F = f(x). \dots . \overline{r}v.\mathbf{0}$$

$$F = f(x) \,.\, g(r) \,.\, \dots . \overline{r}v \,.\, \mathbf{0}$$

- A caller C calls F and waits for the result. There may be several callers.

$$C = \overline{f}u \,.\, r(z). \dots .$$

$$C = \overline{f}u \,.\, (\nu r)\overline{g}r \,.\, r(z) \,.\, \mathbf{0}$$

- How make sure that that only the C who called F will receive the result of its call?

Exercise

- F represent an agent enacting a function. It receives some value along a certain link and produces (a link to) some result:

$$F = f(x). \ldots . \overline{r}v.\mathbf{0}$$

$$F = f(x) \,.\, g(r) \,.\, \ldots . \overline{r}v \,.\, \mathbf{0}$$

- A caller C calls F and waits for the result. There may be several callers.

$$C = \overline{f}u \,.\, r(z). \ldots .$$

$$C = \overline{f}u \,.\, (\nu r)\overline{g}r \,.\, r(z) \,.\, \mathbf{0}$$

- How make sure that that only the C who called F will receive the result of its call?

But do we really know that f gets both x and r from the same C ?

Exercise

- A world with one server S and several clients C.
- S sends two names to any client who will listen. But both names must end up with the same client!

$$S = \overline{a}n_1 \,.\, \overline{a}n_2 \,.\, \mathbf{0}$$
$$C = a(x) \,.\, a(y) \,.\, \ldots$$
$$S \mid C \mid C \cdots \mid C$$

Exercise

- A world with one server S and several clients C.
- S sends <u>two</u> names to any client who will listen. But <u>both</u> names must end up with the <u>same</u> client!

$$S = (\nu p)(\overline{a}p \,.\, \overline{p}n_1 \,.\, \overline{p}n_2 \,.\, \mathbf{0})$$
$$C = a(q) \,.\, q(x) \,.\, q(y) \,.\, \ldots$$
$$S \mid C \mid C \cdots \mid C$$

Additional operators

Sum: P + Q means an agent behaving as either P or Q, resolved at the first action.

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

Same kind of choice as in a nondeterministic automaton: just says that both branches are possible and nothing about how the choice is resolved.

Additional operators

match, mismatch:

$[x = y]P$ means an agent behaving as P if x and y are the same name, ow do nothing.

$[x \neq y]P$ means an agent behaving as P if x and y are not the same name, ow do nothing

$$\frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P', \quad x \neq y}{[x \neq y]P \xrightarrow{\alpha} P'}$$

Additional operators

match, mismatch:

$[x = y]P$ means an agent behaving as P if x and y are the same name, ow do nothing.

$[x \neq y]P$ means an agent behaving as P if x and y are not the same name, ow do nothing

$$\frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P', \quad x \neq y}{[x \neq y]P \xrightarrow{\alpha} P'}$$

$[\varphi]P$ means "if φ then P"

Exercise

An agent receives a name along a. If the received name is u then it is forwarded along b. Otherwise it is forwarded along c.

Exercise

An agent receives a name along a. If the received name is u then it is forwarded along b. Otherwise it is forwarded along c.

$$a(x)\,.\,([x = u]\overline{b}x\,.\,\mathbf{0} + [x \neq u]\overline{c}x\,.\,\mathbf{0})$$

Additional operators

replication and recursion

$!P$ means an agent behaving as an unlimited number of copies of P

$A \Leftarrow P$ means that the agent identifier A represents whatever P is. Note that A can occur in P, leading to recursion.

$$\frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \qquad \frac{A \Leftarrow P, \quad P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}$$

Exercise

Do the following agents behave the same, or else how are they different?

$$A \Leftarrow a(x) \,.\, \overline{b}x \,.\, A$$

$$!a(x) \,.\, \overline{b}x \,.\, \mathbf{0}$$

Exercise

Do the following agents behave the same, or else how are they different?

$A \Leftarrow a(x) \,.\, \overline{b}x \,.\, A$

$A \xrightarrow{au} \overline{b}u \,.\, A$

$!a(x) \,.\, \overline{b}x \,.\, \mathbf{0}$

Exercise

Do the following agents behave the same, or else how are they different?

$$A \Leftarrow a(x)\,.\,\overline{b}x\,.\,A \qquad\qquad A \xrightarrow{au} \overline{b}u\,.\,A$$

$$!a(x)\,.\,\overline{b}x\,.\,\mathbf{0}$$

$$\frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

Exercise

Do the following agents behave the same, or else how are they different?

$$A \Leftarrow a(x) \,.\, \overline{b}x \,.\, A$$

$$A \xrightarrow{au} \overline{b}u \,.\, A$$

$$!a(x) \,.\, \overline{b}x \,.\, \mathbf{0}$$

$$\frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

$$a(x) \,.\, \overline{b}x \,.\, \mathbf{0} \xrightarrow{au} \overline{b}u \,.\, \mathbf{0}$$

$$a(x) \,.\, \overline{b}x \,.\, \mathbf{0} \mid !a(x) \,.\, \overline{b}x \,.\, \mathbf{0} \xrightarrow{au} \overline{b}u \,.\, \mathbf{0} \mid !a(x) \,.\, \overline{b}x \,.\, \mathbf{0}$$

Exercise

Do the following agents behave the same, or else how are they different?

$$A \Leftarrow a(x) \,.\, \overline{b}x \,.\, A \qquad\qquad A \xrightarrow{au} \overline{b}u \,.\, A$$

$$\frac{P \mid\, !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

$$!a(x) \,.\, \overline{b}x \,.\, \mathbf{0}$$

$$a(x) \,.\, \overline{b}x \,.\, \mathbf{0} \xrightarrow{au} \overline{b}u \,.\, \mathbf{0}$$

$$a(x) \,.\, \overline{b}x \,.\, \mathbf{0} \mid\, !a(x) \,.\, \overline{b}x \,.\, \mathbf{0} \xrightarrow{au} \overline{b}u \,.\, \mathbf{0} \mid\, !a(x) \,.\, \overline{b}x \,.\, \mathbf{0}$$

$$!a(x) \,.\, \overline{b}x \,.\, \mathbf{0} \xrightarrow{au} \overline{b}u \,.\, \mathbf{0} \mid\, !a(x) \,.\, \overline{b}x \,.\, \mathbf{0}$$

Exercise

- F represent an agent enacting a function. It receives some value along a certain link and produces (a link to) some result:

$$F = f(x) \,.\, g(r) \,.\, \ldots \overline{r}v \,.\, \mathbf{0}$$

- A caller C calls F and wait for the result. There may be several callers.

- How make sure that that only the C who called F will receive the result of its call?

$$C = \overline{f}u \,.\, (\nu r)\overline{g}r \,.\, r(z) \,.\, \mathbf{0}$$

Exercise

Can we make the function F callable several times?

$$F = f(x) \, . \, g(r) \, . \, \ldots \overline{r}v \, . \, \mathbf{0}$$

Can we make it reentrant?

$$C = \overline{f}u \, . \, (\nu r)\overline{g}r \, . \, r(z) \, . \, \mathbf{0}$$

Exercise

Can we make the function F callable several times?

$$F = f(x) \,.\, g(r) \,.\, \ldots \overline{r}v \,.\, \mathbf{0}$$

$$F \Leftarrow f(x) \,.\, g(r) \,.\, \ldots \overline{r}v \,.\, F$$

Can we make it reentrant?

$$C = \overline{f}u \,.\, (\nu r)\overline{g}r \,.\, r(z) \,.\, \mathbf{0}$$

Exercise

Can we make the function F callable several times?

Can we make it reentrant?

$$F = f(x) \,.\, g(r) \,.\, \ldots \overline{r}v \,.\, \mathbf{0}$$

$$F \Leftarrow f(x) \,.\, g(r) \,.\, \ldots \overline{r}v \,.\, F$$

$$C = \overline{f}u \,.\, (\nu r)\overline{g}r \,.\, r(z) \,.\, \mathbf{0}$$

This disregards that F might receive x and r from different C.

Exercise

Can we make the function F callable several times?

Can we make it reentrant?

$$F = f(x)\,.\,g(r)\,.\,\ldots\,\overline{r}v\,.\,\mathbf{0}$$

$$F \Leftarrow f(x)\,.\,g(r)\,.\,\ldots\,\overline{r}v\,.\,F$$

$$F = !f(x)\,.\,g(r)\,.\,\ldots\,\overline{r}v\,.\,\mathbf{0}$$

$$C = \overline{f}u\,.\,(\nu r)\overline{g}r\,.\,r(z)\,.\,\mathbf{0}$$

This disregards that F might receive x and r from different C.

Bisimulation

When do two agents ”behave the same”?

Bisimulation

When do two agents ”behave the same”?

$\alpha\,.\,\mathbf{0}$ $\qquad\qquad$ $\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}$

Bisimulation

When do two agents ”behave the same”?

$\alpha\,.\,\mathbf{0}$ $\qquad\qquad$ $\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}$

P $\qquad\qquad$ $P + P$

Bisimulation

When do two agents ”behave the same”?

$\alpha \,.\, \mathbf{0}$ $\qquad$ $\alpha \,.\, \mathbf{0} + \alpha \,.\, \mathbf{0}$

P $\qquad$ $P + P$

$\alpha \,.\, \beta \,.\, \mathbf{0} + \beta \,.\, \alpha \,.\, \mathbf{0}$ $\qquad$ $\alpha \,.\, \mathbf{0} \mid \beta \,.\, \mathbf{0}$

Bisimulation

When do two agents "behave the same"?

$\alpha\,.\,\mathbf{0}$	$\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}$
P	$P + P$
$\alpha\,.\,\beta\,.\,\mathbf{0} + \beta\,.\,\alpha\,.\,\mathbf{0}$	$\alpha\,.\,\mathbf{0} \mid \beta\,.\,\mathbf{0}$
$!P$	$!P \mid P$

Bisimulation

When do two agents "behave the same"?

$\alpha \,.\, \mathbf{0}$	$\alpha \,.\, \mathbf{0} + \alpha \,.\, \mathbf{0}$
P	$P + P$
$\alpha \,.\, \beta \,.\, \mathbf{0} + \beta \,.\, \alpha \,.\, \mathbf{0}$	$\alpha \,.\, \mathbf{0} \mid \beta \,.\, \mathbf{0}$
$!P$	$!P \mid P$
$!(P \mid Q)$	$!(P \mid Q) \mid Q$

Bisimulation

When do two agents ”behave the same”?

$\alpha\,.\,\mathbf{0}$	$\alpha\,.\,\mathbf{0}+\alpha\,.\,\mathbf{0}$
P	$P+P$
$\alpha\,.\,\beta\,.\,\mathbf{0}+\beta\,.\,\alpha\,.\,\mathbf{0}$	$\alpha\,.\,\mathbf{0} \mid \beta\,.\,\mathbf{0}$
$!P$	$!P \mid P$
$!(P \mid Q)$	$!(P \mid Q) \mid Q$
$(P \mid Q) \mid R$	$P \mid (Q \mid R)$

Bisimulation

We seek an equivalence relation on agents such that

- Equivalent agents cannot possibly be distinguished in any intuitive way by following transitions
- It is compositional

Bisimulation

First idea: P and Q are equivalent if they have the same actions.

$\alpha\,.\,\mathbf{0}$ $\qquad\qquad$ $\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}$

Bisimulation

First idea: P and Q are equivalent if they have the same actions.

$\alpha\,.\,\mathbf{0}$ $\qquad\qquad$ $\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}$

Not so good, consider

$\beta\,.\,\alpha\,.\,\mathbf{0}$ $\qquad\qquad$ $\beta\,.\,(\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0})$

Bisimulation

$\alpha\,.\,\mathbf{0}$ $\qquad$ $\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}$

Not so good, consider

$\beta\,.\,\alpha\,.\,\mathbf{0}$ $\qquad$ $\beta\,.\,(\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0})$

Bisimulation

Refined idea: P and Q are equivalent if they have the same actions, leading to equivalent agents.

Bisimulation

Refined idea: P and Q are equivalent if they have the same actions, leading to equivalent agents.

Not so good, this definition is circular!

Bisimulation

Not so good, this definition is circular!

Bisimulation

Not so good, this definition is circular!

Can we use induction?

Bisimulation

Inductive definition? Does this work?

P is equivalent to Q if for all α such that $P \xrightarrow{\alpha} P'$ it holds that $Q \xrightarrow{\alpha} Q'$ and P' is equivalent to Q' and vice versa

Bisimulation

Inductive definition? Does this work?

P is equivalent to Q if for all α such that $P \xrightarrow{\alpha} P'$ it holds that $Q \xrightarrow{\alpha} Q'$ and P' is equivalent to Q' and vice versa

We also need $\mathrm{bn}(\alpha) \# Q$

Bisimulation

Inductive definition? Does this work?

P is equivalent to Q if for all α such that $P \xrightarrow{\alpha} P'$ it holds that $Q \xrightarrow{\alpha} Q'$ and P' is equivalent to Q' and vice versa

We also need $\mathrm{bn}(\alpha)\#Q$

But $\xrightarrow{\alpha}$ is not well founded!

Bisimulation

Inductive definition? Does this work?

We also need $\mathrm{bn}(\alpha)\#Q$

But $\xrightarrow{\alpha}$ is not well founded!

Bisimulation

Correct definition is **coinductive** (due to Park 1981)

We seek the largest equivalence relation satisfying

P is equivalent to Q implies that $\forall \alpha . \mathrm{bn}(\alpha) \# Q$ and $P \xrightarrow{\alpha} P'$
it holds that $Q \xrightarrow{\alpha} Q'$ and P' is equivalent to Q'
and vice versa

A relation ~ satisfying this property is called a **bisimulation**

Bisimulation

A relation satisfying this property is not necessarily an equivalence (for example the empty relation is a bisimulation) so a better formulation is

The binary relation R is a *bisimulation* if, for all P, Q:
$P\,R\,Q$ implies that for all α such that $\mathrm{bn}(\alpha)\#Q$ and $P \xrightarrow{\alpha} P'$
it holds that $Q \xrightarrow{\alpha} Q'$ and $P'\,R\,Q'$
and vice versa

The binary relation R is a *bisimulation* if, for all P, Q:

$P\,R\,Q$ implies that for all α such that $\mathrm{bn}(\alpha) \# Q$ and $P \xrightarrow{\alpha} P'$ it holds that $Q \xrightarrow{\alpha} Q'$ and $P'\,R\,Q'$

and vice versa

Three equivalent definitions:

$P \dot{\sim} Q$ if there exists a bisimulation relating P and Q

$\dot{\sim}$ is the union of all bisimulations

$\dot{\sim}$ is the largest bisimulation

$\dot{\sim}$ is called **bisimilarity**


BiSimulation

Examples

$\alpha \,.\, \mathbf{0} \qquad \alpha \,.\, \mathbf{0} + \alpha \,.\, \mathbf{0}$

$P \qquad P + P$

$(P \mid Q) \mid R \qquad P \mid (Q \mid R)$

Examples

$\alpha\,.\,\mathbf{0} \qquad \alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0} \qquad \{(\alpha\,.\,\mathbf{0}, \alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}), (\mathbf{0}, \mathbf{0})\}$

$P \qquad P + P$

$(P \mid Q) \mid R \qquad P \mid (Q \mid R)$

Examples

$\alpha\,.\,\mathbf{0}$ $\qquad$ $\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}$ $\qquad$ $\{(\alpha\,.\,\mathbf{0}, \alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}), (\mathbf{0}, \mathbf{0})\}$

P $\qquad$ $P + P$ $\qquad$ $\{(P, P + P)\} \cup \mathrm{Id}$

$(P \mid Q) \mid R$ $\qquad$ $P \mid (Q \mid R)$

Examples

$\alpha\,.\,\mathbf{0}$ $\quad$ $\alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}$ $\quad$ $\{(\alpha\,.\,\mathbf{0}, \alpha\,.\,\mathbf{0} + \alpha\,.\,\mathbf{0}), (\mathbf{0}, \mathbf{0})\}$

P $\quad$ $P + P$ $\quad$ $\{(P, P + P)\} \cup \mathrm{Id}$

$(P \mid Q) \mid R$ $\quad$ $P \mid (Q \mid R)$ $\quad$ Surprisingly complicated:

$$\{((\nu a_1)\cdots(\nu a_n)((P \mid Q) \mid R), ((\nu a_1)\cdots(\nu a_n)(P \mid (Q \mid R))\cdots\}$$

Compositionality

If $P \dot{\sim} Q$ then

- $P \mid R \dot{\sim} Q \mid R$
- $P + R \dot{\sim} Q + R$
- $!P \dot{\sim} !Q$
- $\overline{a}u \,.\, P \dot{\sim} \overline{a}u \,.\, P$
- $[x = y]P \dot{\sim} [x = y]Q$
- $[x \neq y]P \dot{\sim} [x \neq y]Q$

Compositionality

Does $P \stackrel{\bullet}{\sim} Q$ imply $a(x) \,.\, P \stackrel{\bullet}{\sim} a(x) \,.\, Q$?

Compositionality

Does $P \stackrel{\bullet}{\sim} Q$ imply $a(x) \,.\, P \stackrel{\bullet}{\sim} a(x) \,.\, Q$?

No!

Compositionality

Does $P \stackrel{\bullet}{\sim} Q$ imply $a(x)\,.\,P \stackrel{\bullet}{\sim} a(x)\,.\,Q$?

No!

$$P = \mathbf{0}, \qquad Q = [x = y]\alpha\,.\,\mathbf{0}$$

Compositionality

Does $P \stackrel{\bullet}{\sim} Q$ imply $a(x) \,.\, P \stackrel{\bullet}{\sim} a(x) \,.\, Q$?

No!

$$P = \mathbf{0}, \qquad Q = [x = y]\alpha \,.\, \mathbf{0}$$

Neither has any transition so they are bisimilar.

But

$$a(x) \,.\, Q \xrightarrow{ay} [y = y]\alpha \,.\, \mathbf{0} \xrightarrow{\alpha} \mathbf{0}$$

$$a(x) \,.\, P \xrightarrow{ay} \mathbf{0} \stackrel{\alpha}{\not\longrightarrow}$$

Compositionality

The input prefix $a(x)$ means that x can be substituted by something received

So, for compositionality we also require that agents are bisimilar under substitutions.

$$P \sim Q \text{ if for all } \tilde{x}, \tilde{y} \text{ it holds that } P[\tilde{x} := \tilde{y}] \stackrel{\bullet}{\sim} Q[\tilde{x} := \tilde{y}]$$

Compositionality

~ is a congruence and satisfies several useful laws

There are several versions of bisimilarity. The one presented here is called **strong early bisimilarity**. There are also several alternative ways to present the semantics.

The End for now

- Read in ”An introduction to the pi-calculus”
- Work on the review questions. Tackle those you think you can manage and interest you.
- See you after lunch!