

$x = 1$

let $x = 1$ in ...

$x(1).$

$!x(1)$

$x.set(1)$

Programming Language Theory

Calculus of Communicating Systems

Ralf Lämmel

This lecture is based on a number of different resources as indicated per slide.

Many calculi

Functional programming — Lambda calculus

OO programming — Featherweight Java and others

Concurrency — Calculus of Communicating Systems (CCS) and others

Concurrency

What is concurrency?

What makes **concurrent** programming different from **sequential** programming?

What are the core components of a concurrent language?

Concurrency

- Possible inter-thread communication mechanisms:
 - Read/write to shared memory.
 - Locks.
 - Monitors (a.k.a. wait/notify).
 - Buffered streams.
 - Unbuffered streams.
 - ...
- Which of these does a given language, e.g., Java, support?
- Which should we include in a foundational calculus?

History

- Models of concurrency (late 1970s-80s): Communicating Sequential Processes (Hoare), Petri Nets (Petri), **Calculus of Communicating Systems** (Milner), ...
- Additional features to model dynamic network topologies (late 1980s-90s): **Pi-calculus** (Milner), Higher order pi-calculus (Sangiorgi), Ambients (Cardelli and Gordon), ...

The semantic challenge of concurrency

Program meanings

Program Meanings =
Memories \rightarrow Memories.

Ok for
sequential
programs

Program Meanings =
Memories $\rightarrow \mathcal{P}(\text{Memories})$

Needed for
non-
deterministic
programs

Parallelism and shared memory

Program P_1 : $x := 1 ; x := x + 1$

Program P_2 : $x := 2$

Semantics(P_1) = Semantics(P_2)

Parallelism and shared memory

Program $P_1 : x := 1 ; x := x + 1$

Program $P_2 : x := 2$

Program $Q : x := 3$

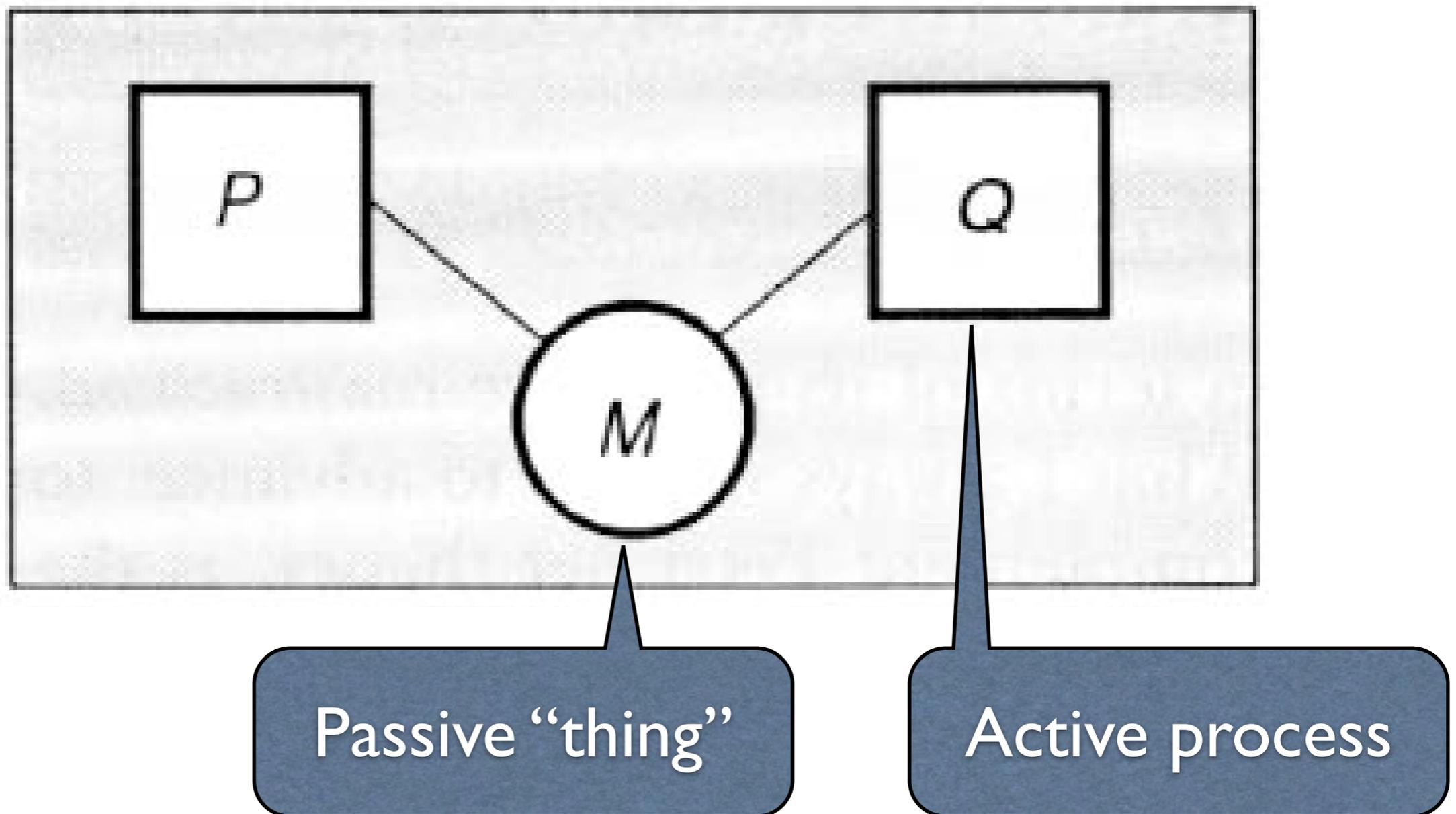
Program $R_1 : P_1 \text{ par } Q$

Program $R_2 : P_2 \text{ par } Q$

Lack of
compositionality

Semantics(R_1) \neq Semantics(R_2)

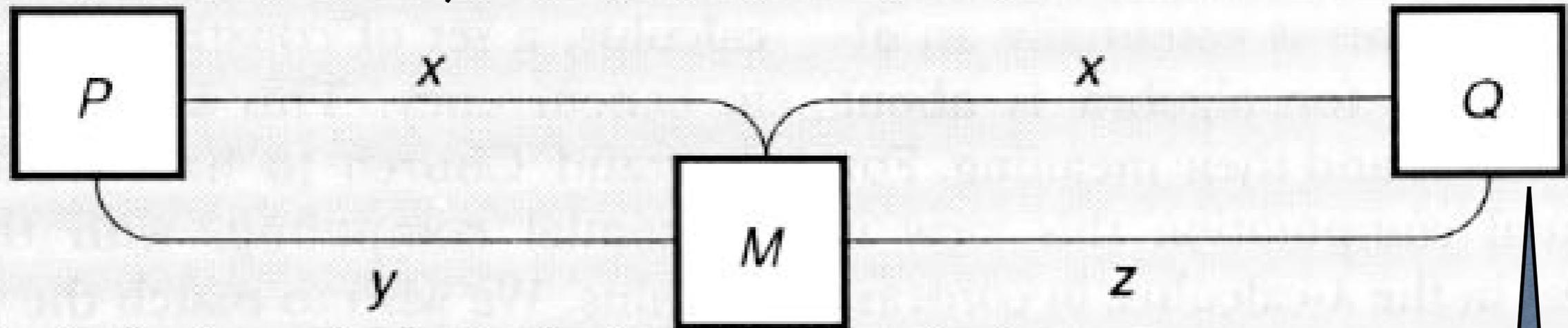
The shared memory model



“Once the memory is no longer at the behest of a single master, then the master-to-slave (or: function-to-value) view of the program-to-memory relationship becomes a bit of a fiction. An old proverb states: He who serves two masters serves none. It is better to develop a general model of interactive systems in which the program-to-memory interaction is just a special case of interaction among peers.”

Memory as an interactive process

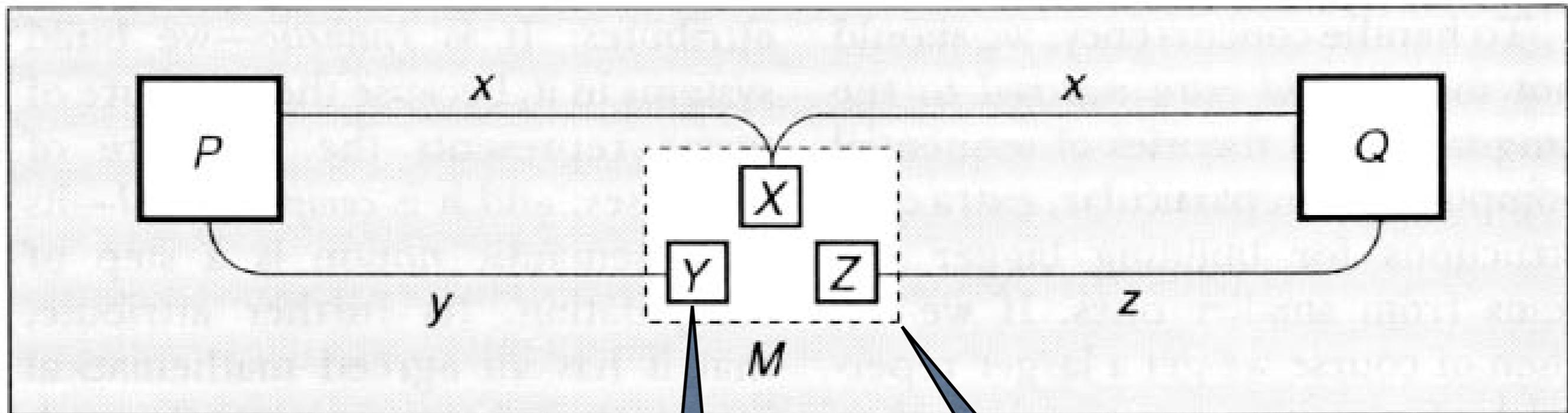
Program variables
as channels



Process

Process

Memory as a distributed process

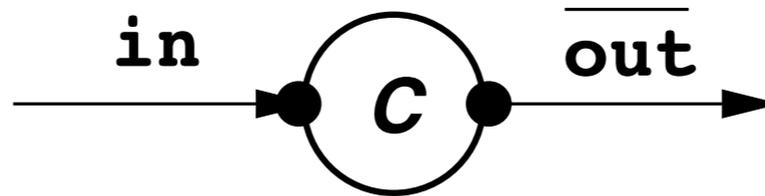


Memory cells are processes.

Memories are no longer monolithic.

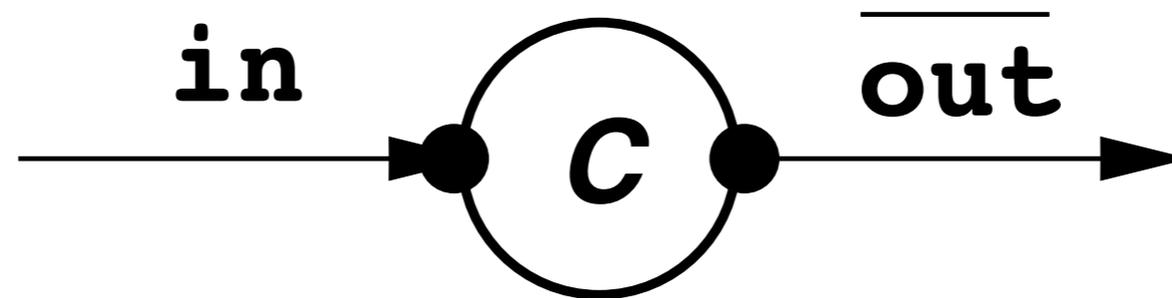
The Calculus of Communicating Systems

Agents and ports



- *Agent C*
 - Dynamic system is network of *agents*.
 - Each agent has own identity persisting over time.
 - Agent performs *actions* (external communications or internal actions).
 - *Behavior* of a system is its (observable) capability of communication.
- Agent has labeled *ports*.
 - Input port *in*.
 - Output port $\overline{\text{out}}$.

A simple example

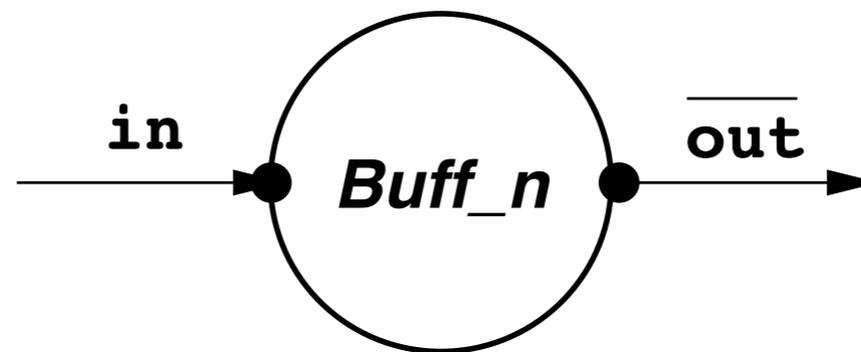


Behavior of C:

- $C := \text{in}(x).C'(x)$
- $C'(x) := \overline{\text{out}}(x).C$

Process behaviors are described as
(mutually recursive) equations.

Example: bounded buffers



Bounded buffer $Buff_n(s)$

- $Buff_n \langle \rangle := in(x).Buff_n \langle x \rangle$
- $Buff_n \langle v_1, \dots, v_n \rangle := \overline{out}(v_n).Buff_n \langle v_1, \dots, v_{n-1} \rangle$
- $Buff_n \langle v_1, \dots, v_k \rangle := in(x).Buff_n \langle x, v_1, \dots, v_k \rangle + \overline{out}(v_k).Buff_n \langle v_1, \dots, v_{k-1} \rangle (0 < k < n)$

Used language elements

- Basic combinator '+'

- $P + Q$ behaves like P or like Q .
- When one performs its first action, other is discarded.
- If both alternatives are allowed, selection is non-deterministic.

- Combining forms

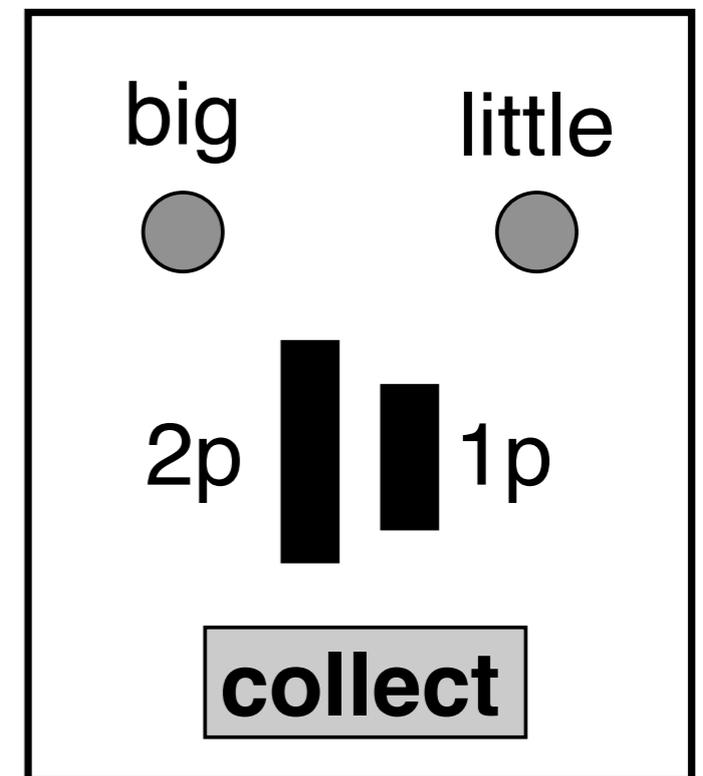
- *Summation* $P + Q$ of two agents.
- *Sequencing* $\alpha.P$ of action α and agent P .

Process definitions may be parameterized.

Later we add
“composition”.

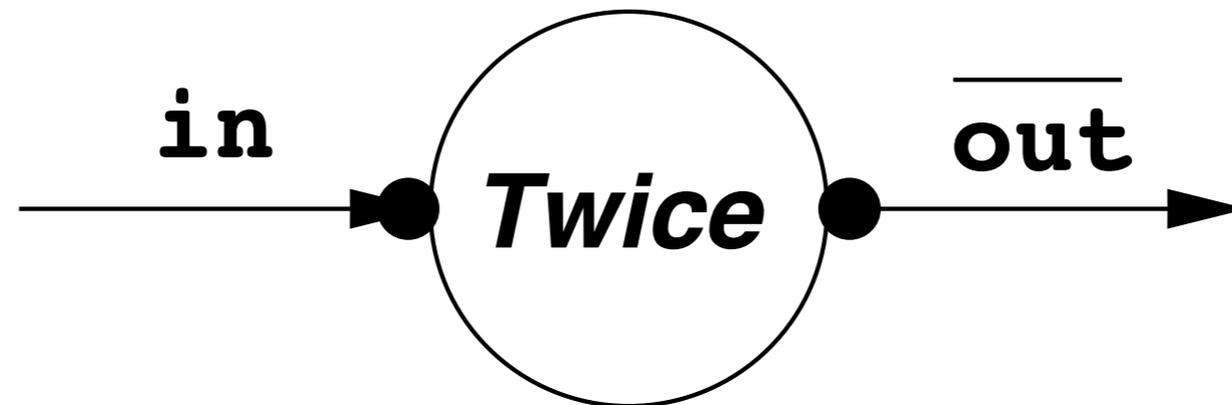
Example: a vending machine

- Big chocolate costs 2p, small one costs 1p.
- $V := 2p.\text{big.collect}.V$
+ $1p.\text{little.collect}.V$



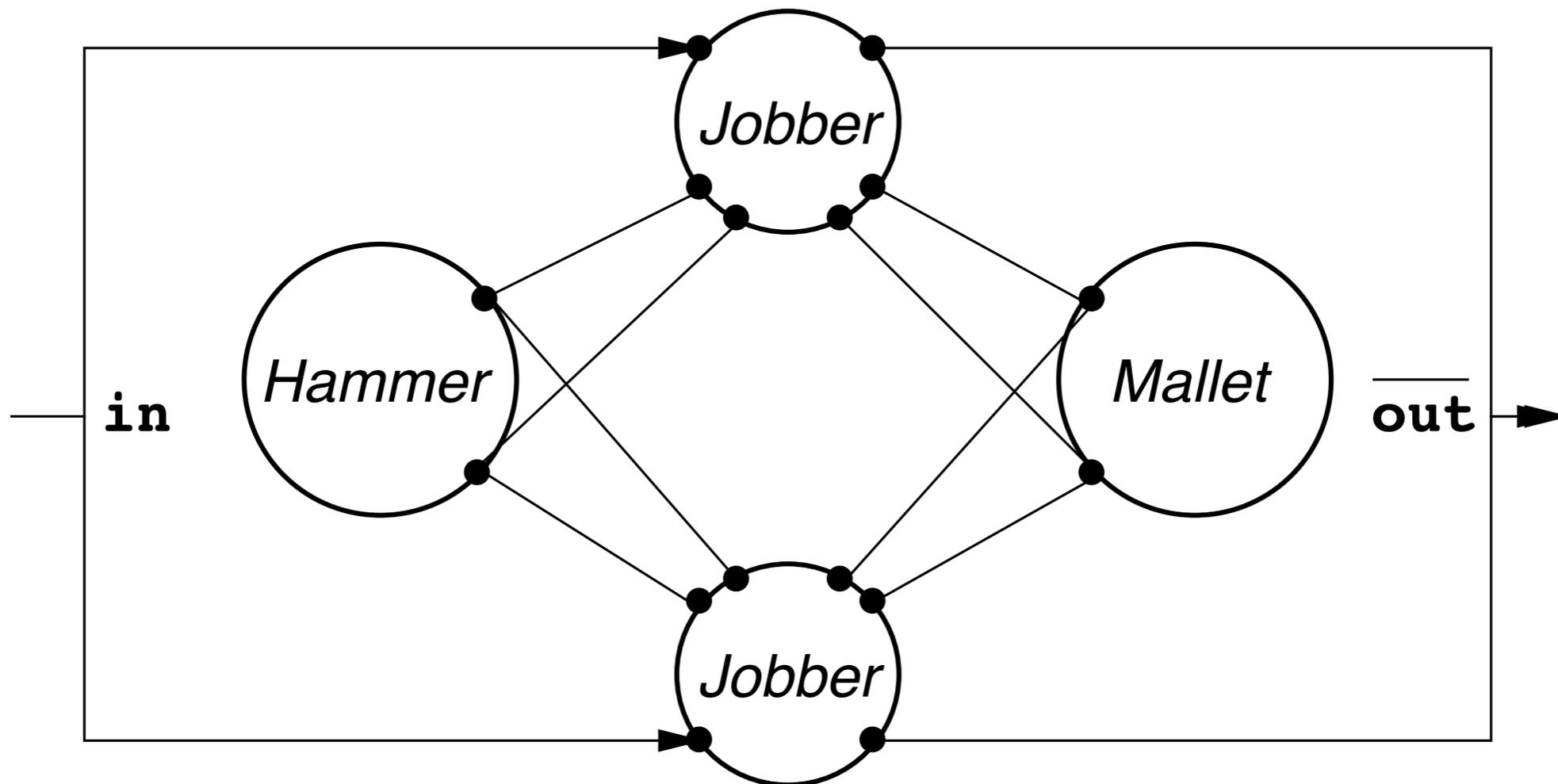
Exercise:
Identify input vs. output.
What behaviors make sense for users?

Example: a multiplier



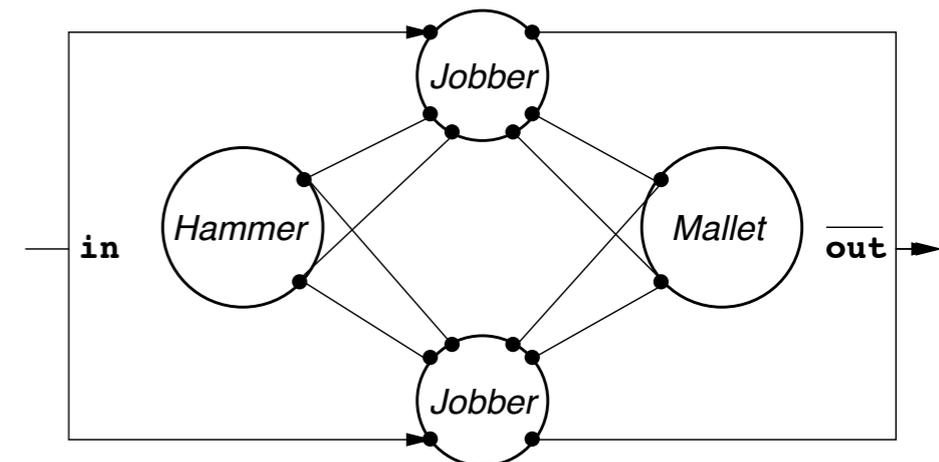
- $Twice := \text{in}(x).\overline{\text{out}}(2 * x).Twice.$
- Output actions may take expressions.

Example: The JobShop



Example: The JobShop

- A simple production line:
 - Two people (the *jobbers*).
 - Two tools (hammer and mallet).
 - *Jobs* arrive sequentially on a belt to be processed.
- Ports may be linked to multiple ports.
 - Jobbers compete for use of hammer.
 - Jobbers compete for use of job.
 - Source of non-determinism.
- Ports of belt are omitted from system.
 - *in* and $\overline{\text{out}}$ are external.
- Internal ports are not labelled:
 - Ports by which jobbers acquire and release tools.



The tools of the JobShop

- Behaviors:

- $\text{Hammer} := \text{geth}.\text{Busyhammer}$

- $\text{Busyhammer} := \text{puth}.\text{Hammer}$

- $\text{Mallet} := \text{getm}.\text{Busymallet}$

- $\text{Busymallet} := \text{putm}.\text{Mallet}$

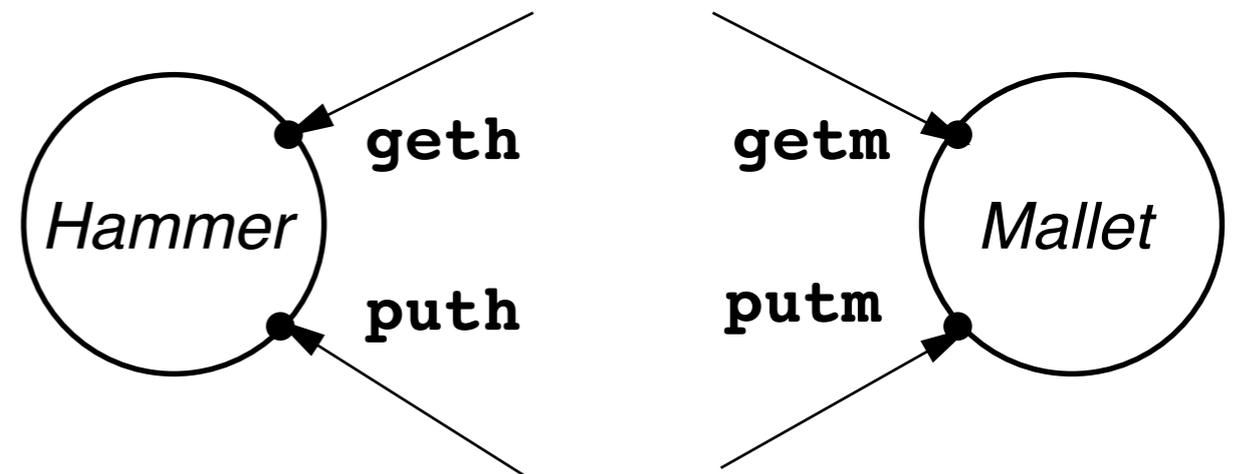
- $\text{Sort} = \text{set of labels}$

- $P : L \dots$ agent P has sort L

- $\text{Hammer}: \{\text{geth}, \text{puth}\}$

- $\text{Mallet}: \{\text{getm}, \text{putm}\}$

- $\text{Jobshop}: \{\text{in}, \overline{\text{out}}\}$



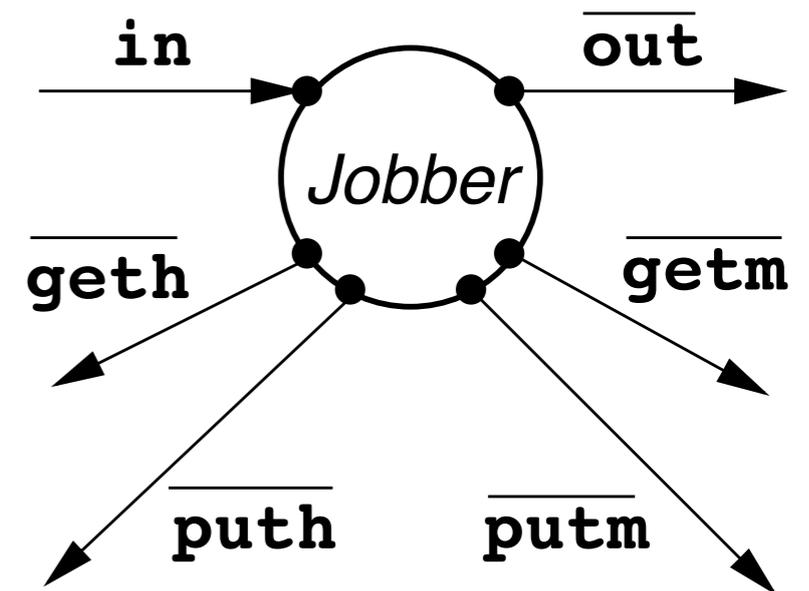
The jobbers of the JobShop

- Different kinds of jobs:

- Easy jobs done with hands.
- Hard jobs done with hammer.
- Other jobs done with hammer or mallet.

- Behavior:

- $Jobber := in(job).Start(job)$
- $Start(job) := \text{if } easy(job) \text{ then } Finish(job)$
else if $hard(job)$ **then** $Uhammer(job)$
else $Usetool(job)$
- $Usetool(job) := Uhammer(job) + Umallet(job)$
- $Uhammer(job) := \overline{geth}.\overline{puth}.Finish(job)$
- $Umallet(job) := \overline{getm}.\overline{putm}.Finish(job)$
- $Finish(job) := \overline{out}(done(job)).Jobber$



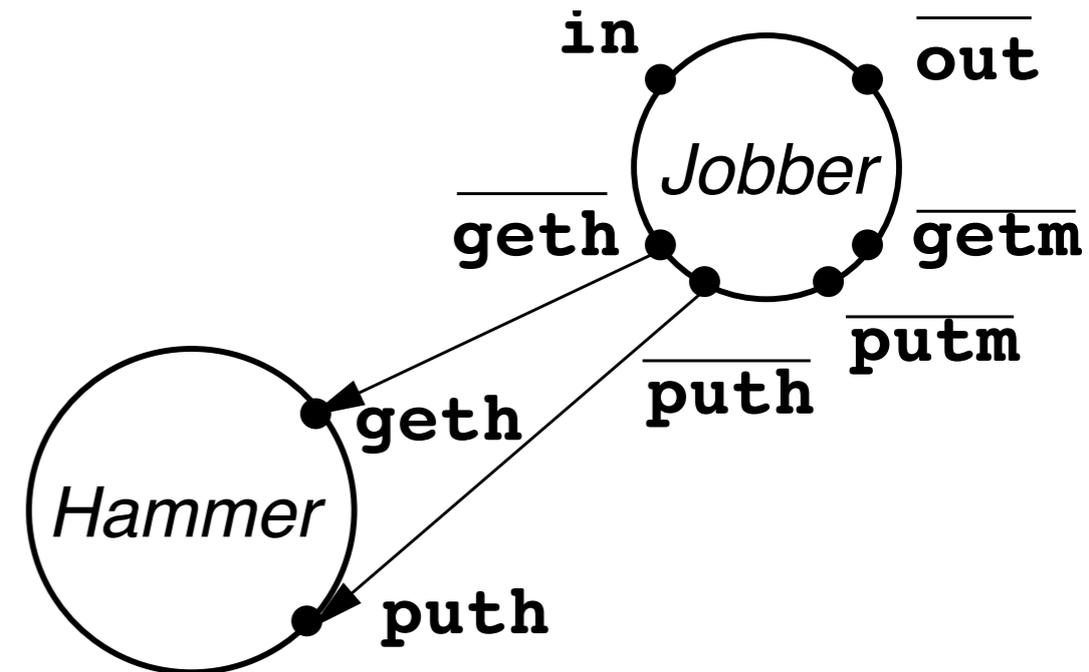
Composition of the agents

- *Jobber-Hammer* subsystem

- *Jobber* | *Hammer*
- *Composition* operator |
- Agents may proceed independently or interact through *complementary* ports.
- Join complementary ports.

- Two jobbers sharing hammer:

- *Jobber* | *Hammer* | *Jobber*
- *Composition* is commutative and associative.



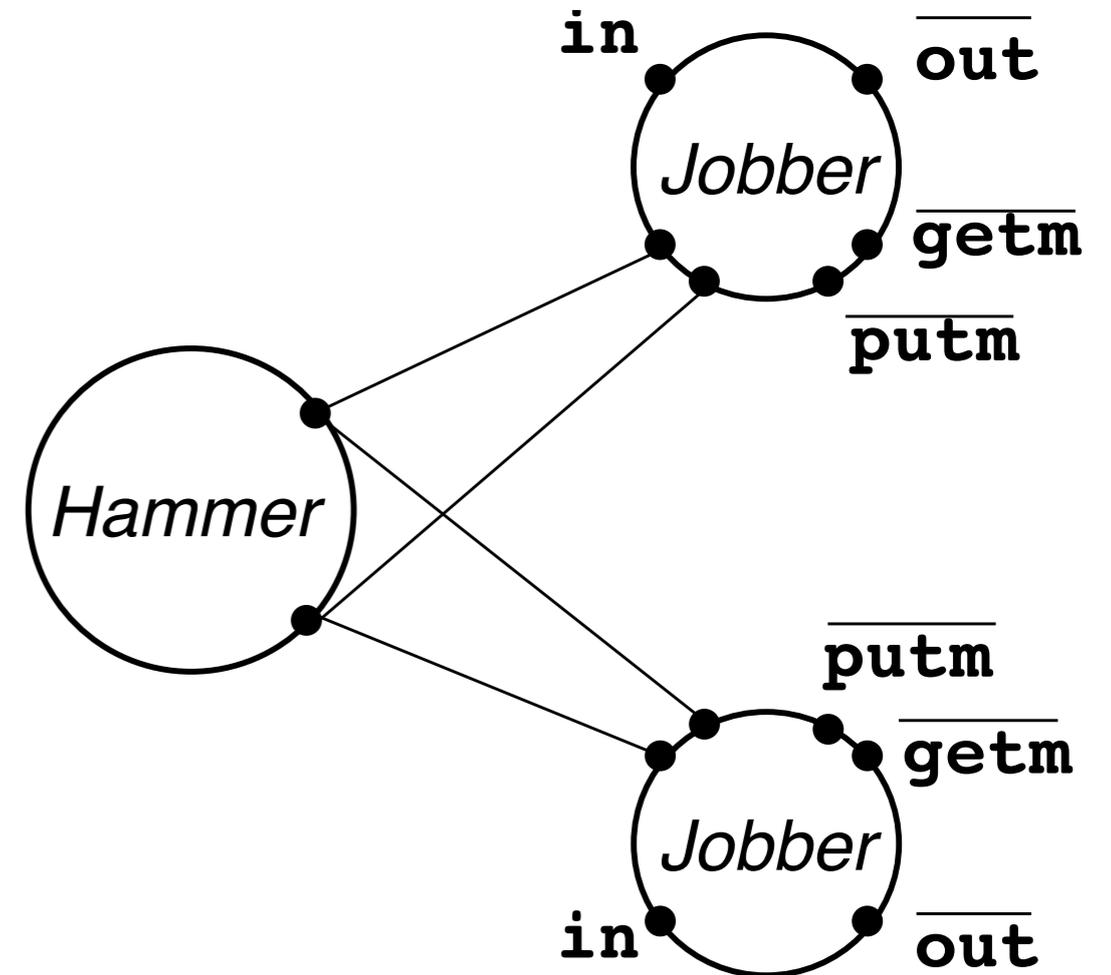
Further composition

- *Internalisation* of ports:

- No further agents may be connected to ports:
- *Restriction* operator \backslash
- $\backslash L$ internalizes all ports L .
- $(Jobber \mid Jobber \mid Hammer) \backslash \{geth, puth\}$

- *Complete system*:

- $Jobshop := (Jobber \mid Jobber \mid Hammer \mid Mallet) \backslash L$
- $L := \{geth, puth, getm, putm\}$



Quote

“... sequential composition is indeed a special case of parallel composition ... in which the only interaction between occurs when P finishes and Q begins ...”

$P; Q$ not part of CCS

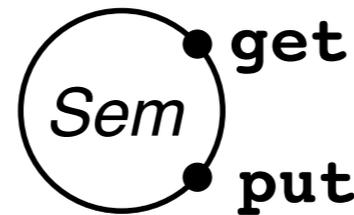
$P|Q$ part of CCS

Reformulations

- *Relabelling Operator*

- $P[l'_1/l_1, \dots, l'_n/l_n]$

- $f(\bar{l}) = \overline{f(l)}$



- Semaphore agent

- $Sem := get.put.Sem$

- Reformulation of tools

- $Hammer := Sem[geth/get, puth/put]$

- $Mallet := Sem[getm/get, putm/put]$



In need of equality of agents

- *Strongjobber* only needs hands:

- *Strongjobber* :=
 $\text{in}(\text{job}).\overline{\text{out}}(\text{done}(\text{job})).\text{Strongjobber}$

- Claim:

- $\text{Jobshop} = \text{Strongjobber} \mid \text{Strongjobber}$
- Specification of system *Jobshop*
- Proof of equality required.



In which sense are the processes equal?

Formalization of CCS

The core calculus

No value transmission: just synchronization

Definitions of agents

- Agent expressions

- Agent constants and variables

- Prefix $\alpha.E$

- Summation ΣE_i

- Composition $E_1|E_2$

- Restriction $E \setminus L$

- Relabelling $E[f]$

Generalization of binary “+”

- Names and co-names

- Set A of *names* (*geth*, *ackin*, ...)

- Set \bar{A} of *co-names* ($\overline{\text{geth}}$, $\overline{\text{ackin}}$, ...)

- Set of *labels* $L = A \cup \bar{A}$

- Actions

- *Completed (perfect) action* τ .

- $\text{Act} = L \cup \{\tau\}$

- Transition $P \xrightarrow{l} Q$ with action l

- $\text{Hammer} \xrightarrow{\text{geth}} \text{Busyhammer}$

Transition rules of the core calculus

- Act $\alpha.E \xrightarrow{\alpha} E$

- Sum_j $\frac{E_j \xrightarrow{\alpha} E'_j}{\Sigma E_i \xrightarrow{\alpha} E'_j}$

- Com₁ $\frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F}$

- Com₂ $\frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$

- Com₃ $\frac{E \xrightarrow{l} E' \quad F \xrightarrow{\bar{l}} F'}{E|F \xrightarrow{\tau} E'|F'}$

This rule rules out transitions with hidden names.

- Res $\frac{E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L} \quad (\alpha, \bar{\alpha} \text{ not in } L)$

- Rel $\frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$

- Con $\frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \quad (A := P)$

This rule makes clear that no more than two agents participate in communication.

This is about the application of definitions for agents.

The value-passing calculus

- Values passed between agents

- Can be reduced to basic calculus.

- $C := \text{in}(x).C'(x)$

- $C'(x) := \overline{\text{out}}(x).C$

- $C := \Sigma_v \text{in}_v.C'_v$

- $C'_v := \overline{\text{out}}_v.C \ (v \in V)$

- Families of ports and agents.

- The full language

- Prefixes $a(x).E, \bar{a}(e).E, \tau.E$

- Conditional **if** b **then** E

- Translation

- $a(x).E \Rightarrow \Sigma_v.E\{v/x\}$

- $\bar{a}(e).E \Rightarrow \bar{a}_e.E$

- $\tau.E \Rightarrow \tau.E$

- **if** b **then** $E \Rightarrow (E, \text{if } b \text{ and } 0, \text{ otherwise})$

Bisimulation (very informally)

Mentioned
in passing

- Two agent expressions P , Q are bisimilar:
 - If P can do an α action towards P' ,
 - then Q can do an α action towards Q' ,
 - such that P' and Q' are again bisimilar,
 - and v.v.

Intuitively two systems are bisimilar if they match each other's moves. In this sense, each of the systems cannot be distinguished from the other by an observer. [Wikipedia]

Laws

Summation laws

$$- P + Q = Q + P$$

$$- P + (Q + R) = (P + Q) + R$$

$$- P + P = P$$

$$- P + 0 = P$$

- Composition laws

- $P|Q = Q|P$

- $P|(Q|R) = (P|Q)|R$

- $P|0 = P$

- Restriction laws

- $P \setminus L = P$, if $L(P) \cap (L \cup \bar{L}) = \emptyset$.

- $P \setminus K \setminus L = P \setminus (K \cup L)$

- ...

- Relabelling laws

- $P[Id] = P$

- $P[f][f'] = P[f' \circ f]$

- ...

Non-laws

- $\tau.P = P$
 - $A = a.A + \tau.b.A$
 - $A' = a.A' + b.A'$
 - A may switch to state in which only b is possible.
 - A' *always* allows a or b .
- $\alpha.(P + Q) = \alpha.P + \alpha.Q$
 - $a.(b.P + c.Q) = a.b.P + a.c.Q$
 - $b.P$ is a -derivative of right side, not capable of c action.
 - a -derivative of left side is capable of c action!
 - Action sequence a, c may yield deadlock for right side.



- **Summary:** *Calculus of Communicating Systems*
 - ♦ *Modeling systems of interacting processes using channels.*
 - ♦ *Approach amenable to formal analysis.*
 - ♦ *Equivalence is based on communication behavior.*
- **Recommended reading:**
 - ♦ *Milner's "Elements of Interaction"*
 - ♦ *CCS tutorial [[AcetoLI05](#)]*