`x = 1`

`let x = 1 in ...`

`x(1).`

`!x(1)`

`x.set(1)`

Programming Language Theory

Concurrency calculi

Ralf Lämmel

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

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 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), ...

In need of designated calculi

Program meanings

Program Meanings =
Memories → Memories.

Ok for sequential programs

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

Ok 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 par Q

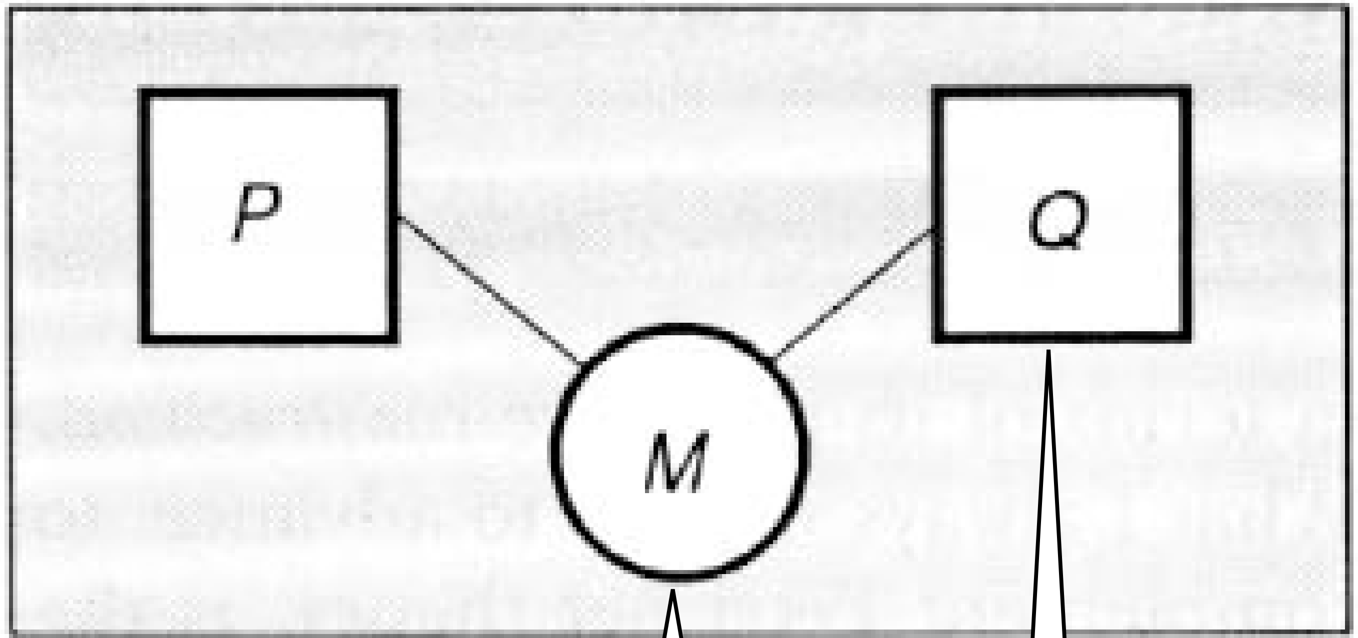
Program R_2 : P_2 par Q

Lack of compositionality

Semantics(R_1) $\neq$ Semantics(R_2)

“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.”

The shared memory model

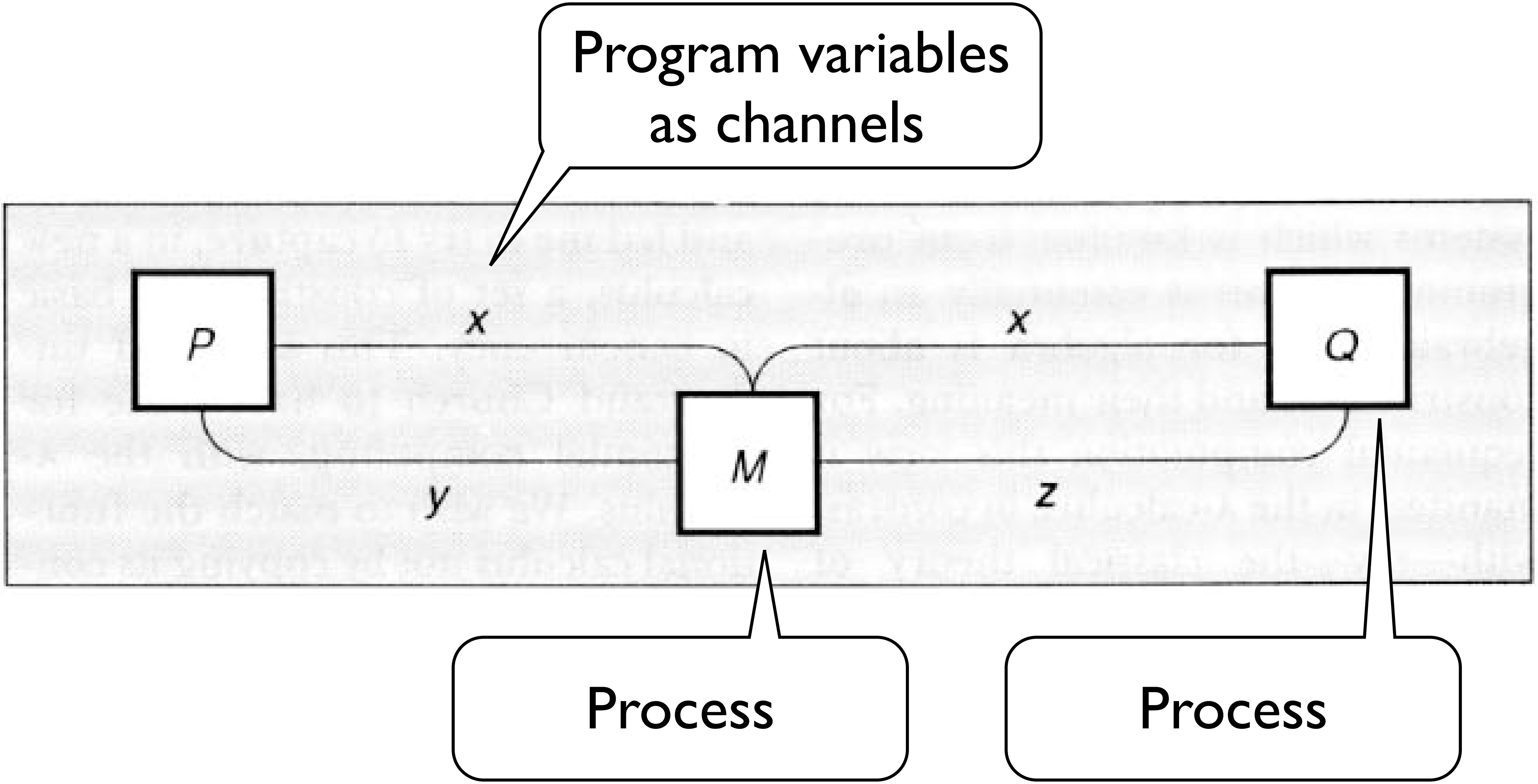


Passive "thing"

Active process

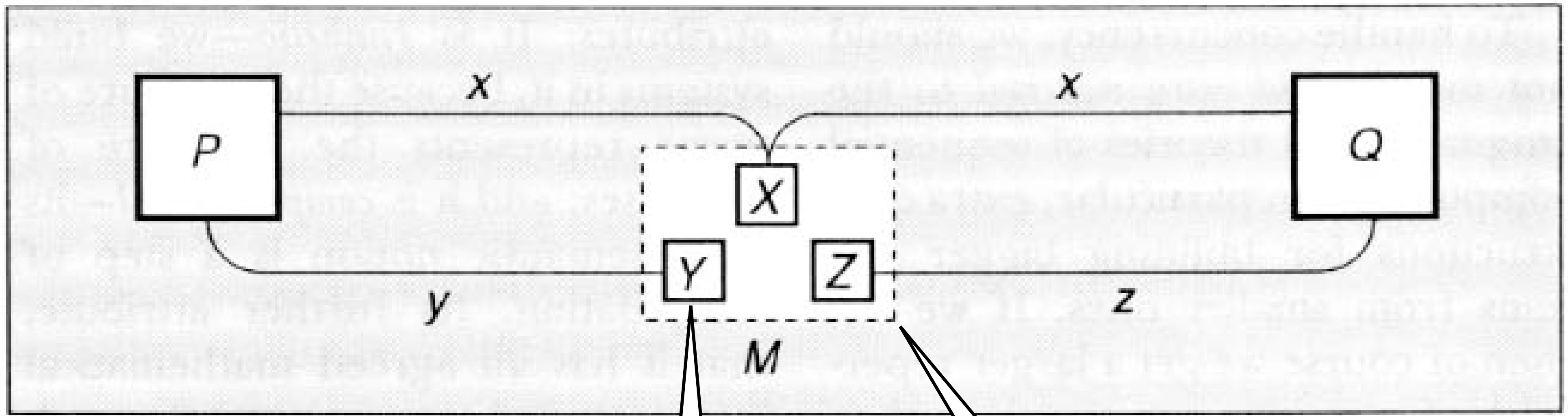
January 1993/Vol.36, No.1/CACM/Robin Milner: "Elements of Interaction"

Memory as an interactive process



January 1993/Vol.36, No.1/CACM/Robin Milner: "Elements of Interaction"

Memory as a distributed process

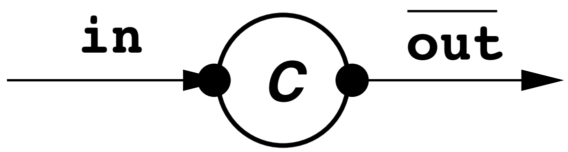


Memory cells are processes.

Memories are no longer monolithic.

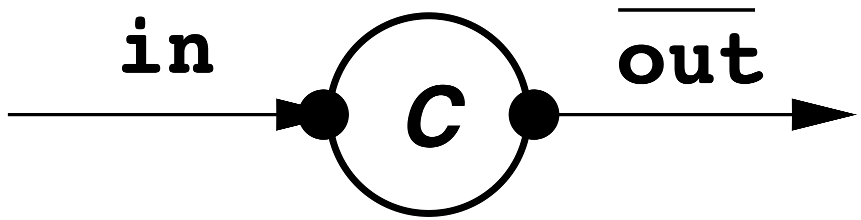
The Calculus of Communicating Systems

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 $\overline{\texttt{out}}$.

A simple example

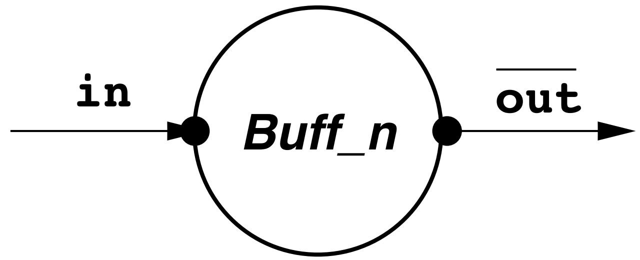


Behavior of C:

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

Process behaviors are described as (mutually recursive) equations.

Example: bounded buffers



Bounded buffer $\textit{Buff}_n(s)$

- $\textit{Buff}_n\ \langle\ \rangle := \texttt{in}(x).\textit{Buff}_n\ \langle x \rangle$
- $\textit{Buff}_n\ \langle v_1, \ldots, v_n \rangle :=$
  $\overline{\texttt{out}}(v_n).\textit{Buff}_n\ \langle v_1, \ldots, v_{n-1} \rangle$
- $\textit{Buff}_n\ \langle v_1, \ldots, v_k \rangle :=$
  $\overline{\texttt{in}}(x).\textit{Buff}_n\ \langle x, v_1, \ldots, v_k \rangle$
  $+\ \overline{\texttt{out}}(v_k).\textit{Buff}_n\ \langle v_1, \ldots, v_{k-1} \rangle (0 < k < n)$

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.

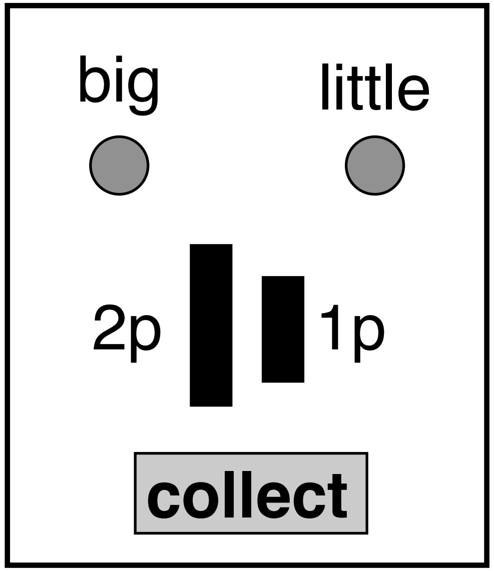
Process definitions may be parameterized.

Later we add "composition".

Example: a vending machine

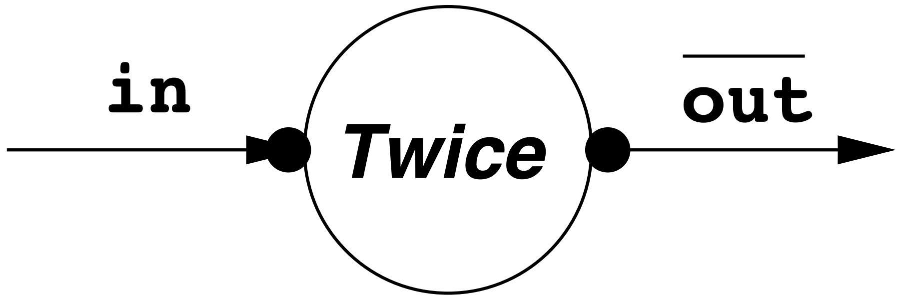
– Big chocolade costs 2p, small one costs 1p.

– $V := \texttt{2p.big.collect.}V$
$\quad + \texttt{1p.little.collect.}V$



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

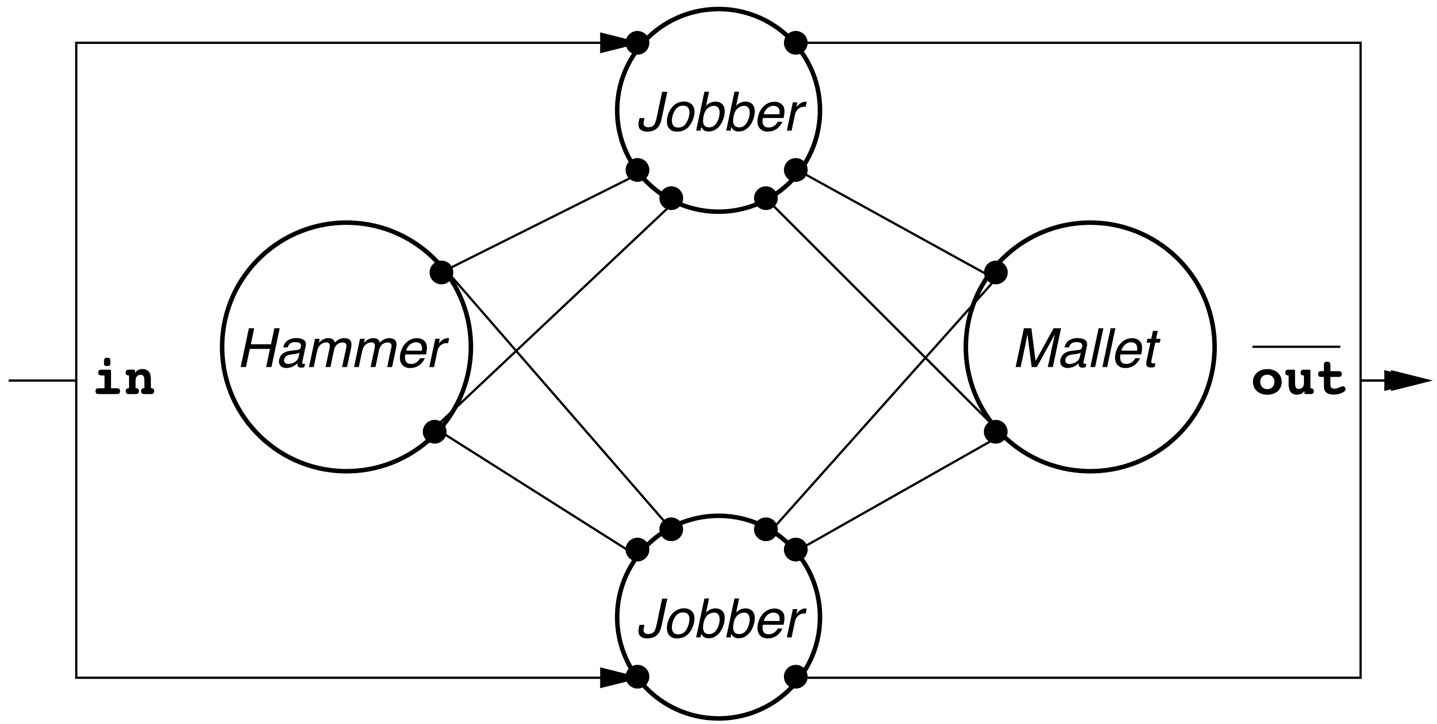
Example: a multiplier



– *Twice* $:= \texttt{in}(x).\overline{\texttt{out}}(2 * x).$*Twice*.

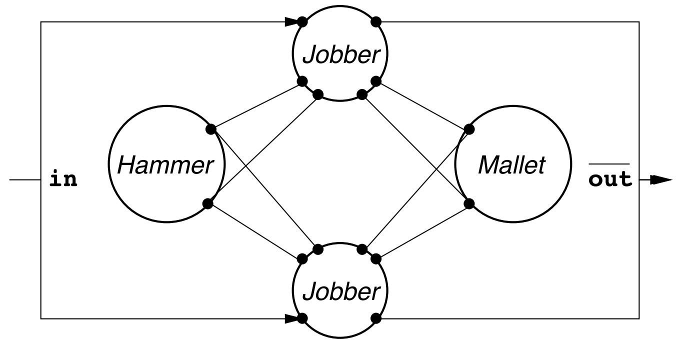
– Output actions may take expressions.

Example: The JobShop



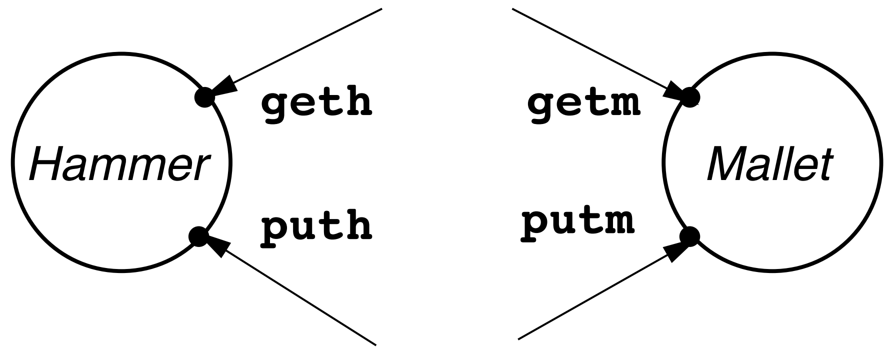
Example: The JobShop

- 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 $\overline{\texttt{out}}$ are external.
- Internal ports are not labelled:
  - Ports by which jobbers acquire and release tools.



The tools of the JobShop

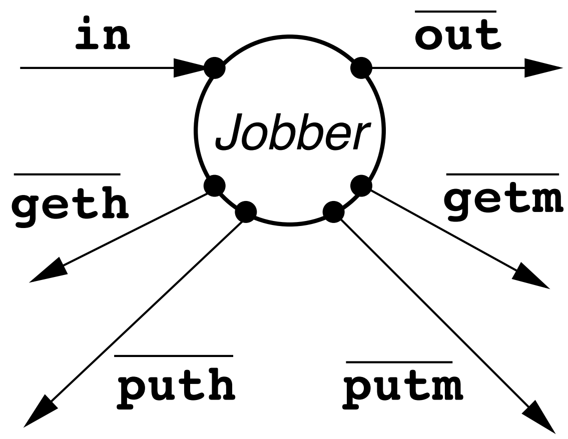
- Behaviors:
  - *Hammer* := *geth*.*Busyhammer*
    Busyhammer := puth.*Hammer*
  - *Mallet* := getm.*Busymallet*
    Busymallet := putm.*Mallet*
- *Sort* = set of labels
  - $P : L$ ... agent P has sort L
  - *Hammer*: {geth, puth}
    Mallet: {getm, putm}
    Jobshop: {in, $\overline{\texttt{out}}$}



These slides were obtained by copy&paste&edit from W. Schreiner's concurrency lectures (Kepler University, Linz).

The jobbers of the JobShop

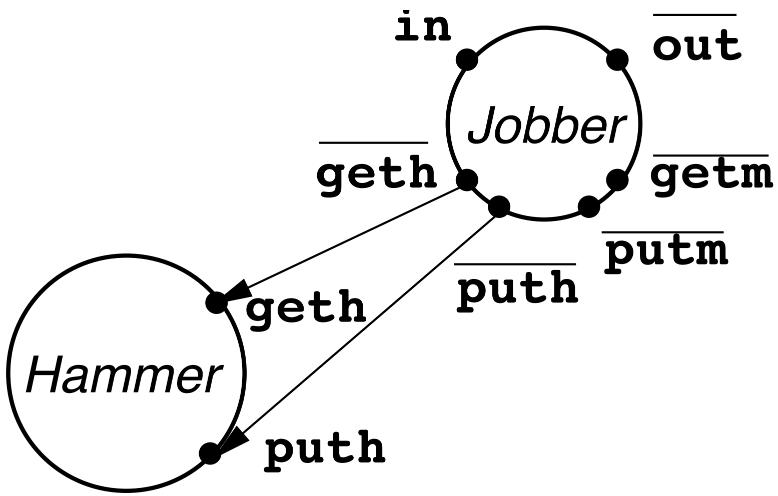
- Different kinds of jobs:
  - Easy jobs done with hands.
  - Hard jobs done with hammer.
  - Other jobs done with hammer or mallet.
- Behavior:
  - $\mathit{Jobber} := \mathtt{in}(\mathit{job}).\mathit{Start}(\mathit{job})$
  - $\mathit{Start}(\mathit{job}) := \textbf{if } \mathit{easy}(\mathit{job}) \textbf{ then } \mathit{Finish}(\mathit{job})$
    $\textbf{else if } \mathit{hard}(\mathit{job}) \textbf{ then } \mathit{Uhammer}(\mathit{job})$
    $\textbf{else } \mathit{Usetool}(\mathit{job})$
  - $\mathit{Usetool}(\mathit{job}) := \mathit{Uhammer}(\mathit{job}) + \mathit{Umallet}(\mathit{job})$
  - $\mathit{Uhammer}(\mathit{job}) := \overline{\mathtt{geth}}.\overline{\mathtt{puth}}.\mathit{Finish}(\mathit{job})$
  - $\mathit{Umallet}(\mathit{job}) := \overline{\mathtt{getm}}.\overline{\mathtt{putm}}.\mathit{Finish}(\mathit{job})$
  - $\mathit{Finish}(\mathit{job}) := \overline{\mathtt{out}}(\mathit{done}(\mathit{job})).\mathit{Jobber}$



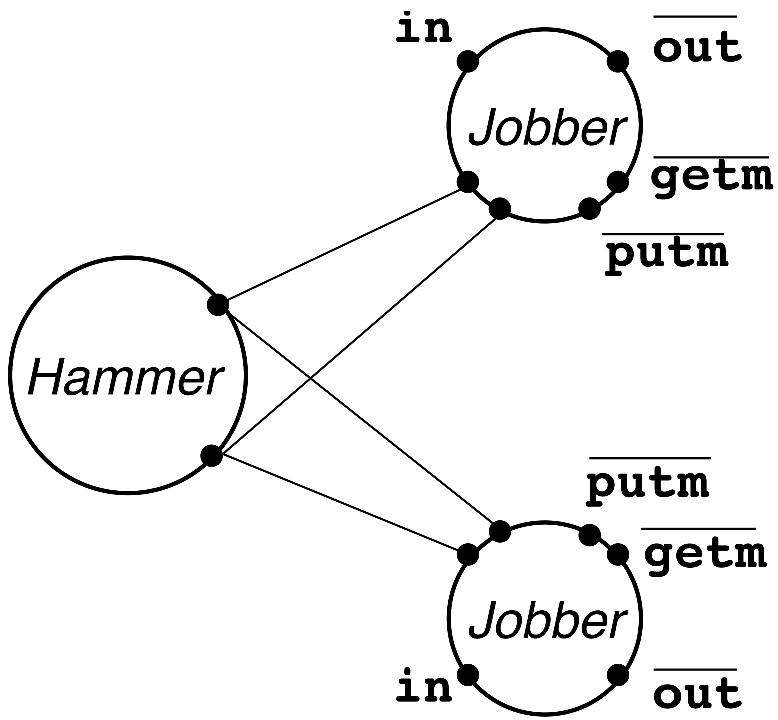
These slides were obtained by copy&paste&edit from W. Schreiner's concurrency lectures (Kepler University, Linz).

Composition of the agents

- *Jobber-Hammer* subsystem
  - *Jobber* | *Hammer*
  - *Composition* operator |
  - Agents may procced 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 \
  - $\backslash L$ internalizes all ports L.
  - (*Jobber* | *Jobber* | *Hammer*)$\backslash\{\texttt{geth},\texttt{puth}\}$
- Complete system:
  - *Jobshop* := (*Jobber* | *Jobber* | *Hammer* | *Mallet*)$\backslash L$
  - $L := \{\texttt{geth},\texttt{puth},\texttt{getm},\texttt{putm}\}$

Quote

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

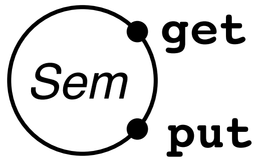
$P; Q$ not part of CCS

$P|Q$ part of CCS

Reformulations

FYI

- *Relabelling* Operator
  - $P[l'_1/l_1,\ldots,l'_n/l_n]$
  - $f(\bar{l}) = \overline{f(l)}$



- 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

FYI

- *Strongjobber* only needs hands:
  - $\mathit{Strongjobber} :=$ $\mathtt{in}(\mathit{job}).\overline{\mathtt{out}}(\mathit{done}(\mathit{job})).\mathit{Strongjobber}$
- Claim:
  - $\mathit{Jobshop} = \mathit{Strongjobber} \mid \mathit{Strongjobber}$
  - *Specification* of system *Jobshop*
  - Proof of equality required.

In which sense are the processes equal?

FYI

Formalization of CCS

Let's skip this
and look at the "simpler" Pi-calculus.

These slides were obtained by copy&paste&edit from W. Schreiner's concurrency lectures (Kepler University, Linz).

The core calculus No value transmission: just synchronization

FYI

Definitions of agents

Generalization of binary "+"

- Agent expressions
  - Agent constants and variables
  - *Prefix* $\alpha.E$
  - *Summation* ΣE_i
  - *Composition* $E_1|E_2$
  - *Restriction* $E\backslash L$
  - *Relabelling* $E[f]$

- Names and co-names
  - Set A of *names* ($\mathtt{geth}$, $\mathtt{ackin}$, ...)
  - Set $\underline{A}$ of *co-names* ($\overline{\mathtt{geth}}$, $\overline{\mathtt{ackin}}$, ...)
  - Set of *labels* $L = A \cup \overline{A}$
- Actions
  - *Completed (perfect)* action τ.
  - *Act* $= L \cup \{\tau\}$
- Transition $P \xrightarrow{l} Q$ with action l
  - *Hammer* $\overset{\mathtt{geth}}{\rightarrow}$ *Busyhammer*

Transition rules of the core calculus

FYI

- Act $\alpha.E \xrightarrow{\alpha} E$
- Sum$_j$ $\dfrac{E_j \xrightarrow{\alpha} E_j'}{\Sigma E_i \xrightarrow{\alpha} E_j'}$
- Com$_1$ $\dfrac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F}$
- Com$_2$ $\dfrac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$
- Com$_3$ $\dfrac{E \xrightarrow{l} E' \quad F \xrightarrow{\overline{l}} F'}{E|F \xrightarrow{\tau} E'|F'}$

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

This rule rules out transitions with hidden names.

- Res $\dfrac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L}$ (α, $\overline{\alpha}$ not in L)
- Rel $\dfrac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$
- Con $\dfrac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}$ $(A := P)$

This is about the application of definitions for agents.

The value-passing calculus

FYI

- Values passed between agents
  - Can be reduced to basic calculus.
  - $C := \mathtt{in}(x).C'(x)$
    $C'(x) := \overline{\mathtt{out}}(x).C$
  - $C := \Sigma_v\ \mathtt{in}_v.C'_v$
    $C'_v := \overline{\mathtt{out}}_v.C\ (v \in V)$
  - *Families* of ports and agents.
- The full language
  - *Prefixes* $a(x).E$, $\overline{a}(e).E$, $\tau.E$
  - *Conditional* **if** b **then** E
- Translation
  - $a(x).E \Rightarrow \Sigma_v.E\{v/x\}$
  - $\overline{a}(e).E \Rightarrow \overline{a}_e.E$
  - $\tau.E \Rightarrow \tau.E$
  - **if** b **then** $E \Rightarrow$ (E, if b and 0, otherwise)

Bisimulation (very informally)

FYI

- Two agent expressions *P*, *Q* are bisimular:
  - 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 bisimular,
  - 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]

FYI

Laws

FYI

Summation laws

- $P + Q = Q + P$
- $P + (Q + R) = (P + Q) + R$
- $P + P = P$
- $P + 0 = P$

FYI

- Composition laws
  - $P|Q = Q|P$
  - $P|(Q|R) = (P|Q)|R$
  - $P|0 = P$
- Restriction laws
  - $P\backslash L = P$, if $L(P) \cap (L \cup \overline{L}) = \emptyset$.
  - $P\backslash K\backslash L = P\backslash(K \cup L)$
  - $\ldots$
- Relabelling laws
  - $P[\mathit{Id}] = P$
  - $P[f][f'] = P[f' \circ f]$
  - $\ldots$

Non-laws

FYI

- $\tau.P = P$
  - $A = a.A + \tau.b.A$
  - $A' = a.A' + b.A'$
  - 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.

Pi-calculus

A minimal model with 'enough stuff' to perform interesting computation (e.g. is more powerful than the lambda-calculus).

Pi calculus

http://en.wikipedia.org/wiki/%CE%A0-calculus

First shot:

$P, Q, R ::=$
$\mathbf{0}$
out $x\ y$; P
in $x\ (y)$; P
$P \mid Q$

Completed process

Output prefixing:
emit name *y* on channel *x*

Input prefixing:
wait for a name on channel *x* to be bound to *y*

Concurrency

Example programs

1. out *stdout hello*; out *stdout world*; **0**
2. in *stdin* (*name*); out *stdout hello*; out *stdout name*; **0**
3. (out *c fred*; **0**) | (in *c* (*name*); out *d name*; **0**)
4. (out *c fred*; out *c wilma*; **0**) | (in *c* (*x*); out *d x*; **0**) | (in *c* (*y*); out *e y*; **0**)
5. (out *c fred*; in *d x*; **0**) | (in *c* (*y*); out *d wilma*; **0**)
6. (in *d x*; out *c fred*; **0**) | (in *c* (*y*); out *d wilma*; **0**)
7. (out *c fred*; in *d* (*x*); **0**) | (out *d wilma*; in *c* (*y*); **0**)

What do these programs do?

Dynamic semantics

Structural congruence $P \equiv Q$ is generated by:

1. If $P =_\alpha Q$ then $P \equiv Q$.
2. $P \mid Q \equiv Q \mid P$.
3. $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$.

Dynamic semantics $P \rightarrow Q$ is generated by:

1. $(\text{out } x\ y; P) \mid (\text{in } x\ (z); Q) \rightarrow \text{P} \mid \text{Q}[y/z]$
2. If $P \rightarrow Q$ then $P \mid R \rightarrow Q \mid R$.
3. If $P \equiv \rightarrow \equiv Q$ then $P \rightarrow Q$.

Recursion? Looping? Infinite Behavior?

Minimal solution *replication*: $!P$ ‘acts like’ $P \mid P \mid P \mid \ldots$

Examples:

1. !in x (z); out y z; **0**
2. out *acquire lock*; **0** | !in *release* (*lock*); out *acquire lock*; **0**

Replicated input !in *accept* (*socket*); P acts a lot like a multithreaded server (Java ServerSocket).

Dynamic semantics just given by:

$$!P \equiv P \mid !P$$

Creation of new channels

Minimal solution *channel generation*: new (x); P generates a fresh channel for use in P.

Example:

1. new (c); out x c; in c (y_1); .. in c (y_n); P
2. in x (c); out c z_1; .. out c z_n; Q

Put these in parallel, and what happens?

New channel generation acts a lot like new object generation / new key generation / new nonce generation / ...

Dynamic semantics just given by:

(new (x); P) | $Q \equiv$ new (x); (P | Q) (as long as $x \notin Q$)

If $P \rightarrow Q$ then new (x); $P \rightarrow$ new (x); Q.

Derived forms

FYI

Multiple messages:

in x $(y_1,\ldots,y_n)$; P
= new (c); out x c; in c (y_1); .. in c (y_n); P

out x $(z_1,\ldots,z_n)$; Q
= in x (c); out c z_1; .. out c z_n; Q

Let's double check:

(in x $(y_1,\ldots,y_n)$; P | out x $(z_1,\ldots,z_n)$; Q) $\rightarrow^*$
$P[z_1/y_1,\ldots,z_n/y_n]$ | Q

In need of garbage collection

FYI

$$\text{new } (c);\ P =_{gc} P \qquad (\text{when } c \notin P)$$

$$\text{new } (c);\ \text{in } c\ (x);\ P =_{gc} \mathbf{0}$$

$$\text{new } (c);\ !\text{in } c\ (x);\ P =_{gc} \mathbf{0}$$

$$\text{new } (c);\ \text{out } c\ x;\ P =_{gc} \mathbf{0}$$

$$\text{new } (c);\ !\text{out } c\ x;\ P =_{gc} \mathbf{0}$$

$$P \mid \mathbf{0} =_{gc} P$$

Let's double check:

$$(\ \text{in } x\ (y_1,\ldots,y_n);\ P \mid \text{out } x\ (z_1,\ldots,z_n);\ Q\ )$$
$$\rightarrow^{*} =_{gc} P[z_1/y_1,\ldots,z_n/y_n] \mid Q$$

Correctness of GC

FYI

Correctness of garbage collection:

If $P =_{gc} Q$ and $P \rightarrow P'$
then $P' =_{gc} Q'$ and $Q \rightarrow Q'$

More derived forms

FYI

Booleans:

True(b)
 = !in b (x, y); out x (); **0**

False(b)
 = !in b (x, y); out y (); **0**

if (b) { P } else { Q }
 = new (t); new (f); (out b (t, f); **0** | in t (); P | in f (); Q)

Sanity check:

True(b) | if (b) { P } else { Q }
 $\longrightarrow^{*} =_{gc}$ True(b) | P

Many derived forms

Can also code integers, linked lists, ...

and the lambda-calculus...

and concurrency controls like mutexes, mvars, ivars, buffers, etc.

- **Summary**: *CCS and Pi-calculus*
  - *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]
- **Outlook**:
  - End Prolog-driven section of this course
  - Begin Haskell-driven section
  - (Preparation of) Midterm