

A *simple, distributed*
implementation of the pi-calculus,
using *explicit fusions*

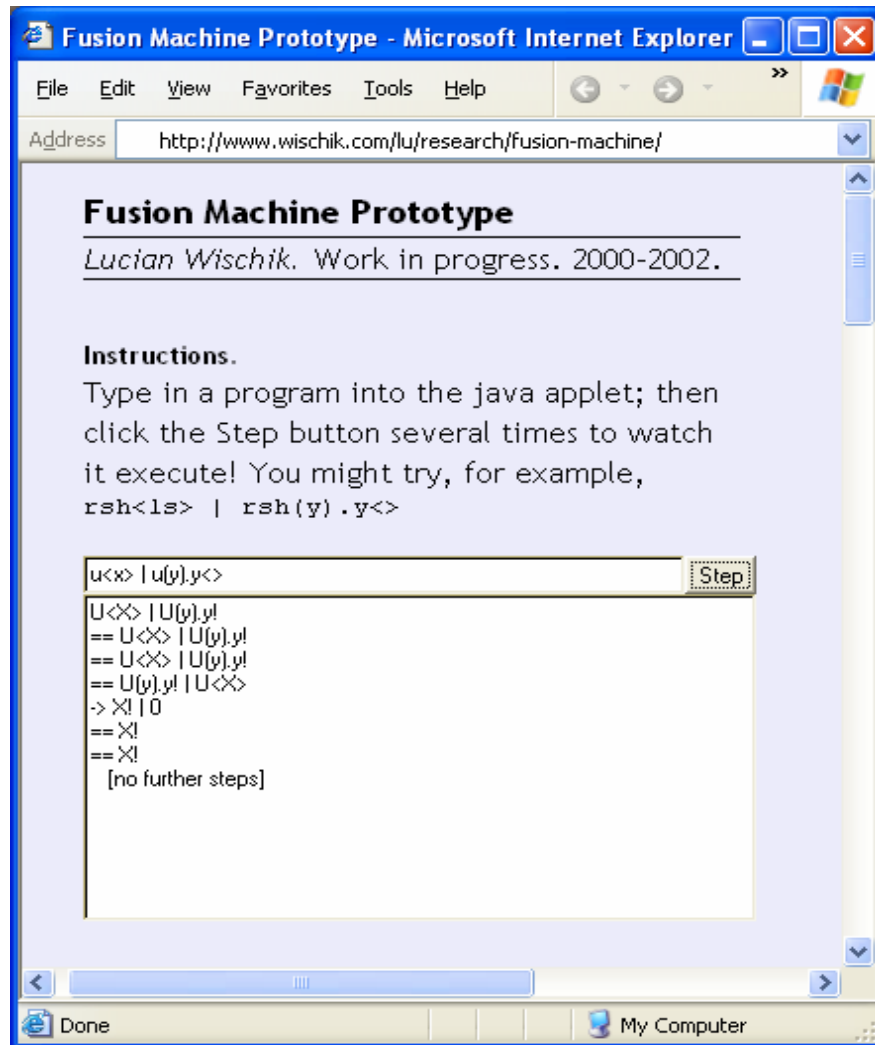
Pisa, July 2002

Lucian Wischik

and

Cosimo Laneve, Philippa Gardner

Manuel Mazzara, Lorenzo Agostinelli



- Paper at Concur 2002
www.wischik.com/lu/research/
- Online prototype
(see left)
- Implementations in Jocaml,
Prolog by Bologna students
- This is the start of an
implementation project

the pi calculus, e.g.

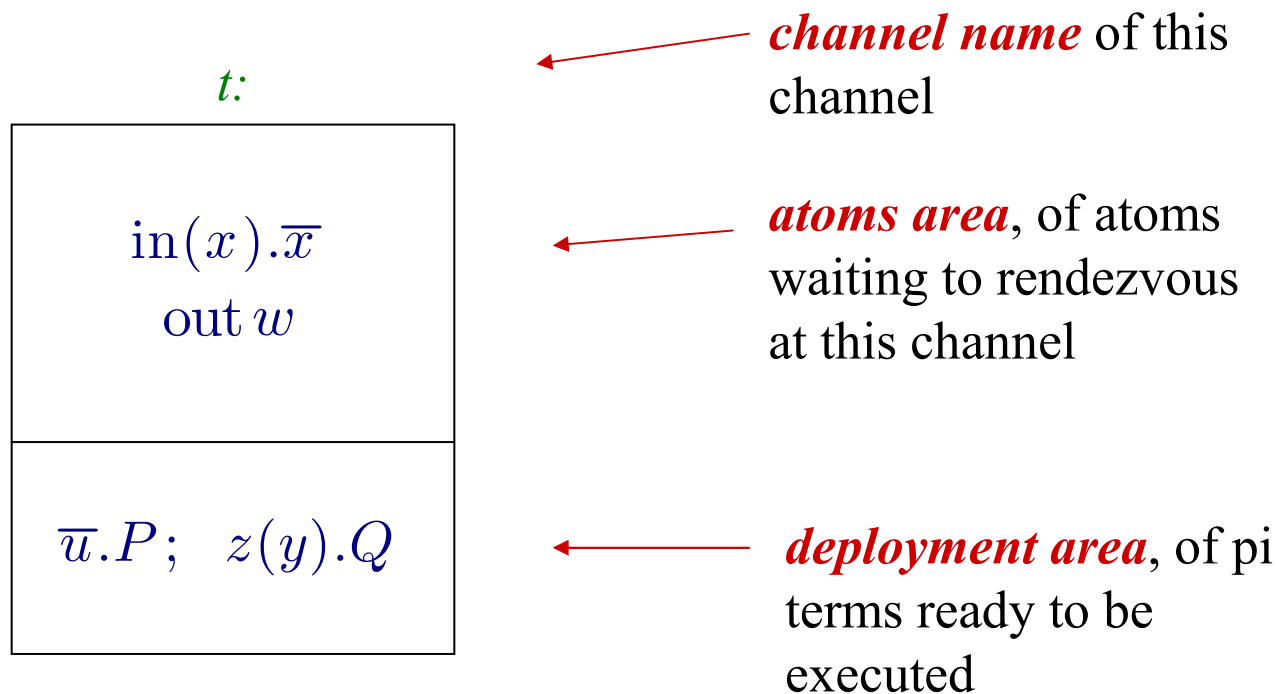
```
(new  $\overline{tunnel}$  @pisa) (  
   $\overline{tunnel}$  wischik com  
  |  $tunnel(x).\overline{x}$   
)  
|  $wischik\ com.\overline{alert}$  // create a fresh channel, at pisa  
// send data 'wischik' over it  
// receive portname, then send on it  
// when we receive an (empty) msg, alert
```

Questions about distribution:

Where is the stuff located on the network?

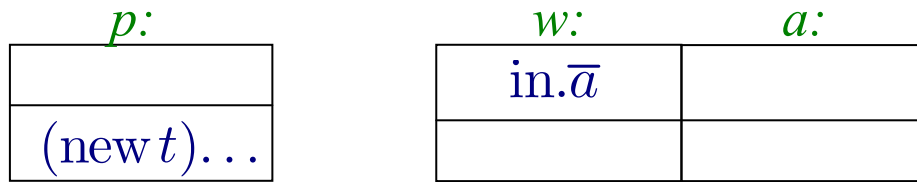
How ***efficiently*** does it run?

distributed channel machine

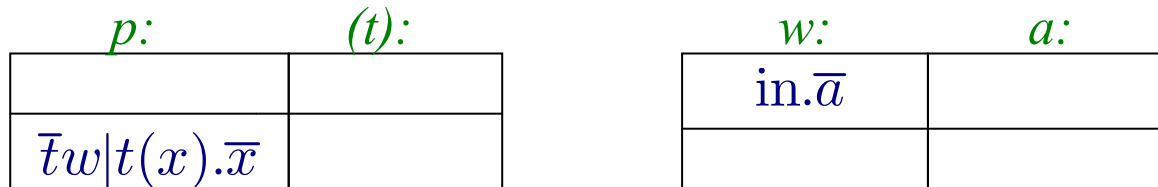


The system is composed *only* of a collection of these distributed channel machines.

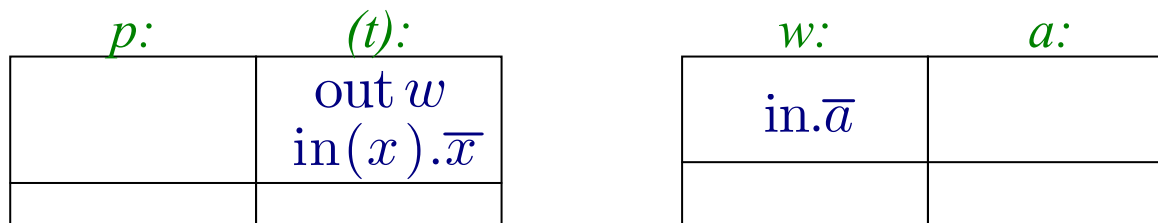
This one corresponds to $t(x).\bar{x} \mid \bar{t}w \mid \bar{u}.P \mid z(y).Q$

$$(\text{new } t@p)(\bar{t}w \mid t(x).\bar{x}) \mid w.\bar{a}$$


create a new channel, co-located with pisa (i.e. execute the “new” command)



Deploy the input & output atoms to their appropriate queue



Reaction! A matching input and output at the same channel can react together

$p:$	$(t):$
	\bar{w}

$w:$	$a:$
$\text{in.}\bar{a}$	



*again, deploy at atom to appropriate location
(by sending it over the network)*

$p:$	$(t):$

$w:$	$a:$
$\text{in.}\bar{a}$ out	



react

$p:$	$(t):$

$w:$	$a:$
\bar{a}	



deploy

$p:$	$(t):$

$w:$	$a:$
	out

... also, garbage-collect (t)

virtual machine, formally

$$\begin{array}{lcl}
 u[\text{out } x.P; \text{in } (y).Q] & \xrightarrow{\text{react}} & u[P; Q\{x/y\}] \\
 u[\text{out } x.P; !\text{in } (y).Q] & \xrightarrow{\text{react}} & u[P; Q\{x/y\}; !\text{in } (y).Q] \\
 \\
 u[\bar{v} x.P] \ v[] & \xrightarrow{\text{dep.out}} & u[] \ v[\text{out } x.P] \\
 u[(\text{new } x)P] & \xrightarrow{\text{dep.new}} & u[P\{x'/x\}] \ (x')[] \quad * \\
 \\
 u[P|Q] & \xrightarrow{\text{dep.par}} & u[P; Q] \\
 \\
 u[\mathbf{0}] & \xrightarrow{\text{dep.nil}} & u[]
 \end{array}$$

* x' fresh, unique

THEOREM $P \sim Q$ iff $u[P] \sim u[Q]$

main problem

$$u(x).v(y).w(z).P \mid \bar{u}a \mid \bar{v}b \mid \bar{w}c$$

Q. Example will transport all of P first to u , then v , then w .
How to implement this more efficiently?

?A. Guard P and then, at the last minute, transport P direct to its final destination. (Parrow, 1999). But this causes a latency problem...

$$(\text{new } t) (u(x).v(y).w(z).\bar{t}xyz \mid t(xyz).P)$$

main problem

$$u(x).v(y).w(z).P \mid \bar{u}a \mid \bar{v}b \mid \bar{w}c$$

Q. Example will transport all of P first to u , then v , then w .
How to implement this more efficiently?

A. Optimistically send P to its expected final destination. Use *explicit fusions* (Gardner and Wischik, 2000). Then, if we had sent it to the wrong place, it will become *fused* to the correct place and it can *migrate*...

the explicit fusion calculus

$$P ::= x=y \mid \bar{u}\tilde{x}.P \mid u\tilde{x}.P \mid P|P \mid (x)P \mid \mathbf{0}$$

$$\bar{u}\tilde{x}.P \mid u\tilde{y}.Q \longrightarrow \tilde{x}=\tilde{y} \mid P \mid Q$$

$$x=y \mid P \equiv x=y \mid P\{y/x\} \quad \textit{substitution}$$

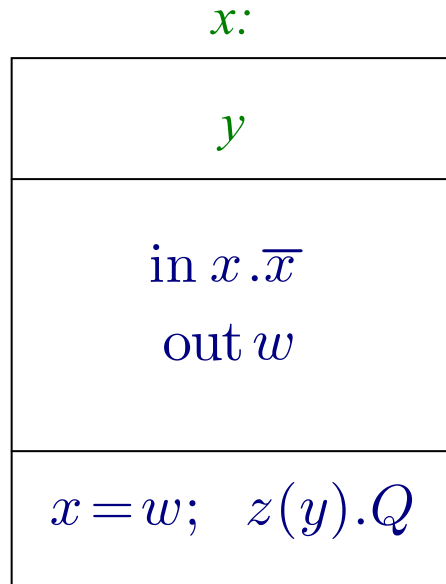
$$(x)(x=y) \equiv \mathbf{0} \quad \textit{local alias}$$

$$x=x \equiv \mathbf{0} \quad \textit{reflexivity}$$

$$x=y \equiv y=x \quad \textit{symmetry}$$

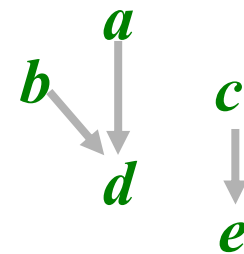
$$x=y \mid y=z \equiv x=z \mid y=z \quad \textit{transitivity}$$

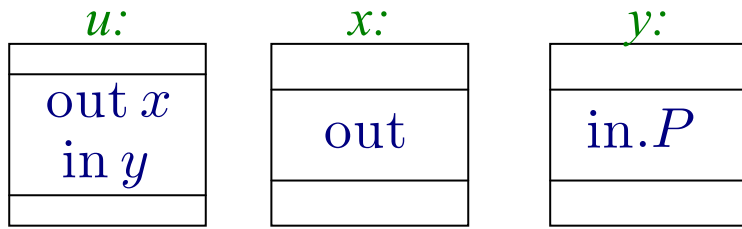
fusion machine



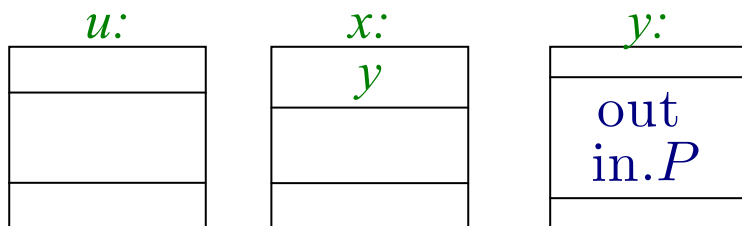
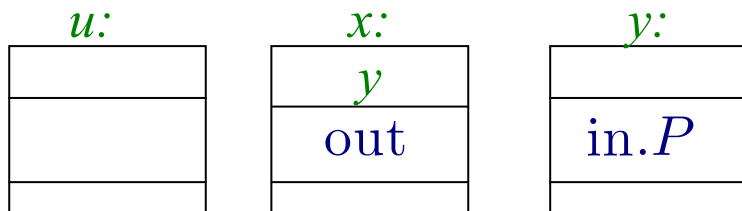
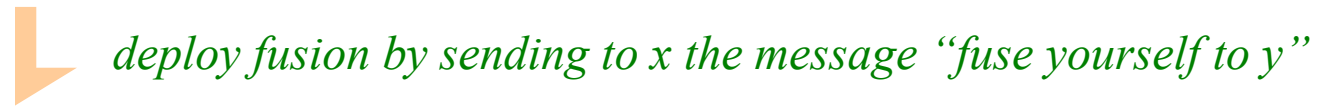
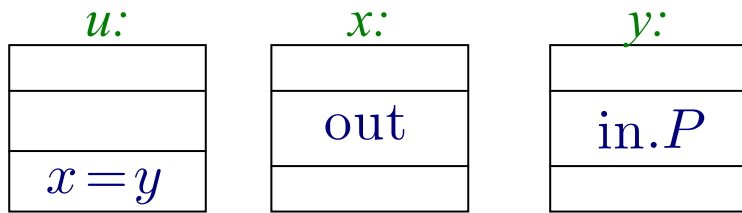
← *fusion pointer*, so any atom can migrate from here to y .

Collectively, the fusion pointers make a *forest* which respects a total order on names:





$$\begin{aligned} \bar{u}x \mid uy \mid \bar{x} \mid y.P &\longrightarrow x=y \mid \bar{x} \mid y.P \\ &\longrightarrow x=y \mid P \end{aligned}$$



THEOREM

$$P \sim Q \text{ iff } u[P] \sim u[Q]$$

fusion results

THEOREM

- Using explicit fusions, we can compile a program with continuations into one without.
- This is a source-code optimisation, prior to execution.
- Every message becomes small (fixed-size).
- This might double the total number of messages but no worse than that. It also reduces latency.
- Our optimisation *is* a bisimulation congruence:

$$C[P] \sim C[\text{optimise } P]$$

```
(new xyz, v'@v, w'@w) (  
  ux. v'=v           // after u has reacted, it tells  
  | v'y. w'=w        // v' to fuse to v, so allowing  
  | w'z              // our v' atom to react with v atoms  
)
```

what we are discovering

THOUGHTS

- Channel-based makes for *easy implementation*.
(I have implemented it in java and C++.
Students have implemented it in Jocaml and Prolog).
Also makes for *easy and strong proofs of correctness*.
- Fusions allow for *optimisation* at source level, by
“pre-deploying” fragments to their expected destination.
- The machine is just a start. Substantial work needed to build a
full implementation and *language* on top of it...
XML data types (Mazzara, Meredith).
Transactions and rollbacks like Xlang. This is motivated by the
problem of ‘false fusions’ like $2=3$, and seems the best way to
deal with failure (Laneve, Wischik, Meredith).
Quantify the cost of fusion/migration.

Supplemental Slides

Grammar for fusion machine calculus

Implementation notes

Fusion algorithm

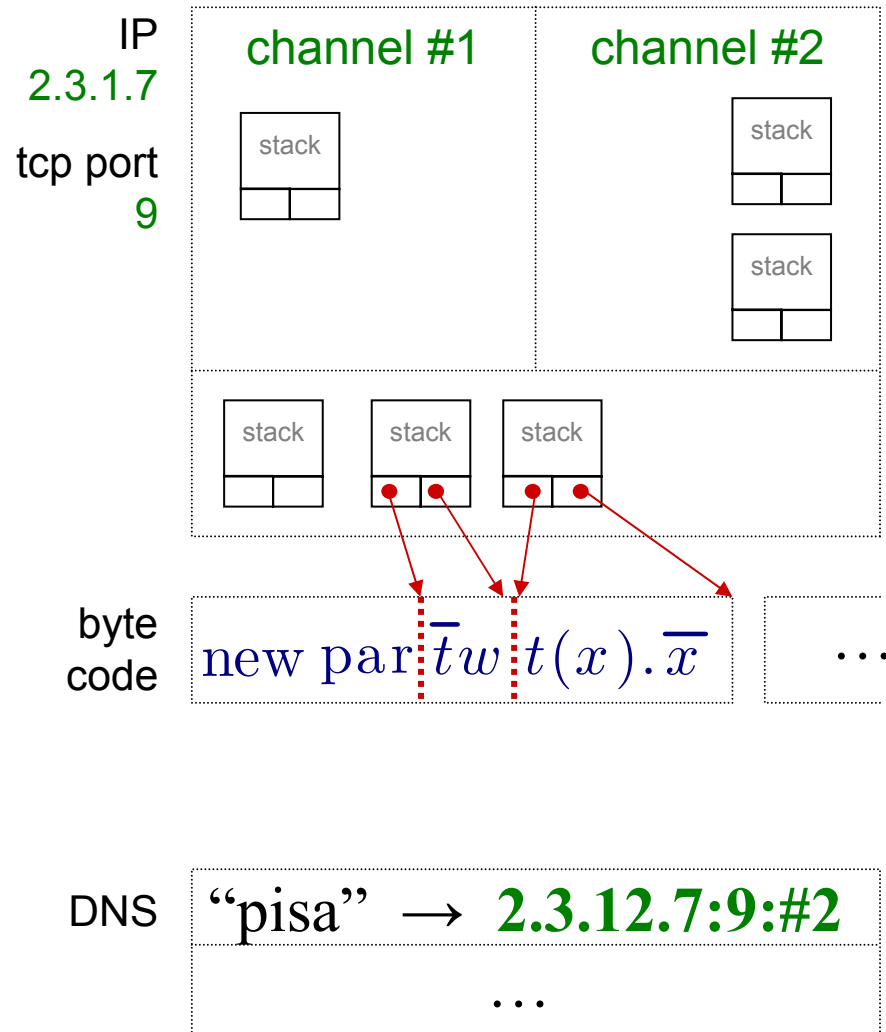
virtual machine, formally


Machines M ::= $u[B]$ *channel machine at u*
 $(u)[B]$ *private channel machine*
 M, M
 $\mathbf{0}$



Bodies B ::= $\text{out}\tilde{x}.P$ *output atom*
 $\text{in}(\tilde{x}).P$ *input atom*
 $!\text{in}(\tilde{x}).P$ *replicated input*
 P *pi process*
 $B; B$

Processes P ::= $\bar{u}\tilde{x}.P$ | $![!u(\tilde{x}).P$ | $(x)P$ | $P|P$ | $\mathbf{0}$

virtual machine in practice



Server thread:
accepts incoming **work units**  over the network

- Worker threads:
- pick up a work unit  from the “work bag”
 - if it's **PAR**, spawn another 
 - if it's a **remote in/out**, send over network
 - if it's a **local in/out**, either react or add to channel's queue

plan: integrate with C++

Treat functions as addresses

- a name $n = 2.3.1.7 : 9 : 0x04367110$
- so that `snd(n)` will invoke the function at that address

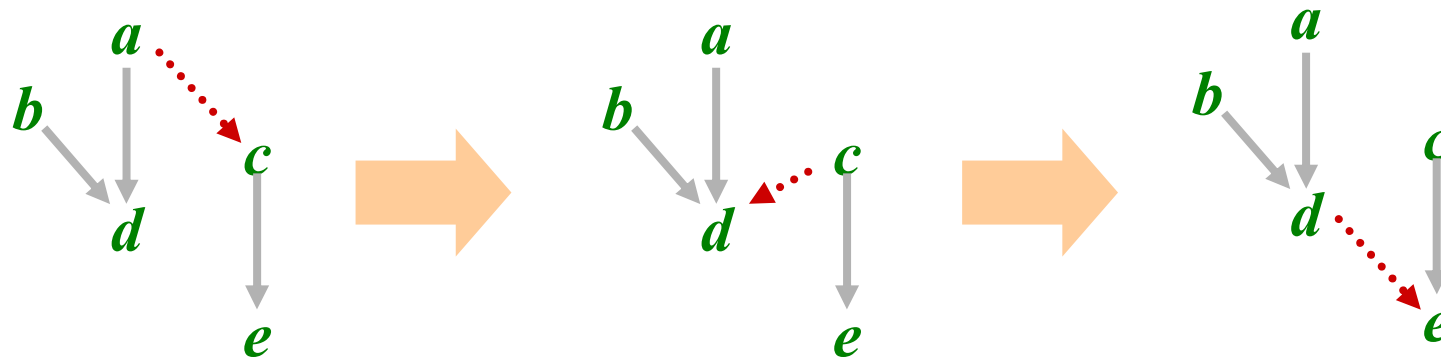
Calling `snd/rcv` directly from C++

```
{ ...           // there's an implicit continuation K after the rcv,  
  rcv(x) ;     // so we stall the thread and put  $x.K$  in the work bag.  
  ...           // When K is invoked, it signals the thread to wake up  
}
```

Calling arbitrary pi code from C++

```
pi ("u!x.v!y | Q") ;  
pi ("u!x." + fun_as_chan(&test2) + "|Q") ;  
  
void test2()  
{ ... }
```

fusion merging results



Effect: a distributed, asynchronous algorithm for merging trees.

- Correctness: it preserves the total-order on channels names;
- the equivalence relation on channels is preserved, before and after;
- it terminates, since each step moves closer to the root.

(similar to Tarjan's *Union Find* algorithm, 1975)

fusion merging algorithm

$$u[x=y] \ x[p:] \xrightarrow{\text{dep.fu}} u[] \ x[y:y=p]$$

* assuming $x < y$
* if p was nil, then
discard $y=p$ in the result

The explicit fusion $x=y$ is an *obligation to set up a fusion pointer*.

A channel will either fulfil this obligation (if p was nil), or will pass it on.