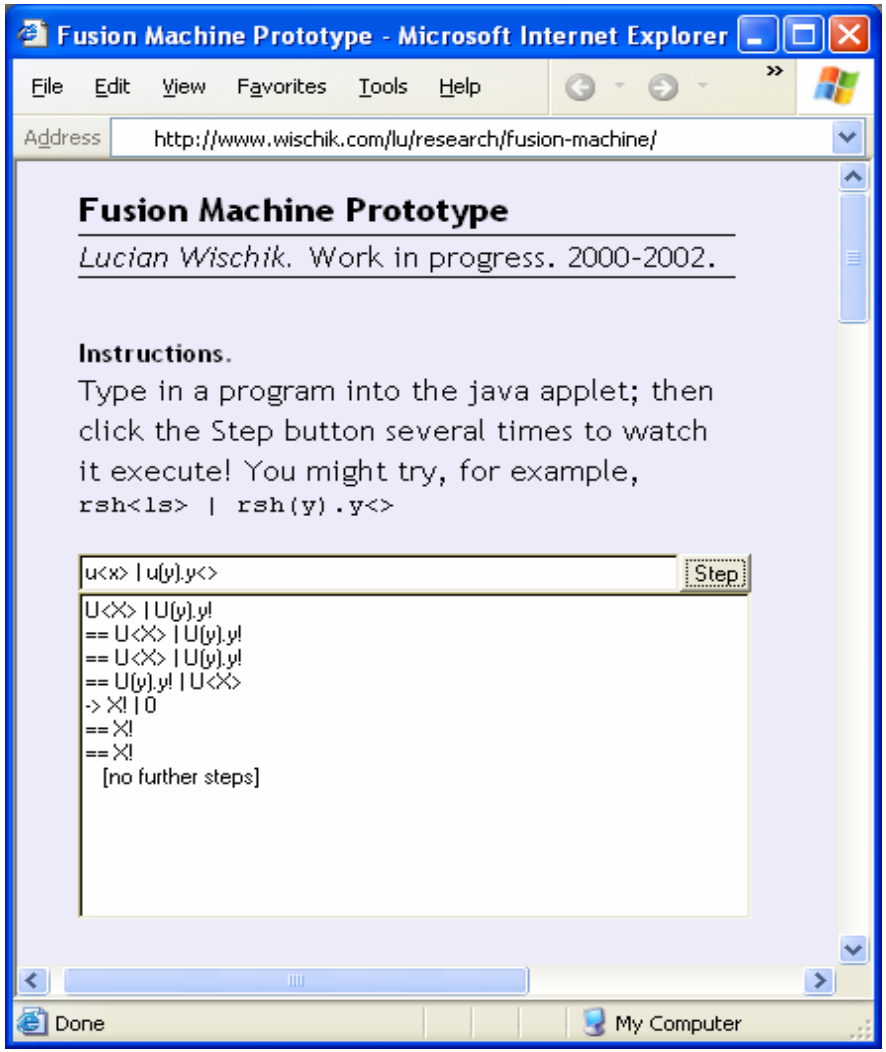
A ***simple***, ***distributed*** implementation of the pi-calculus, using ***explicit fusions***

Pisa, July 2002

Lucian Wischik
and
Cosimo Laneve, Philippa Gardner
Manuel Mazzara, Lorenzo Agostinelli



- Paper at Concur 2002
  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.

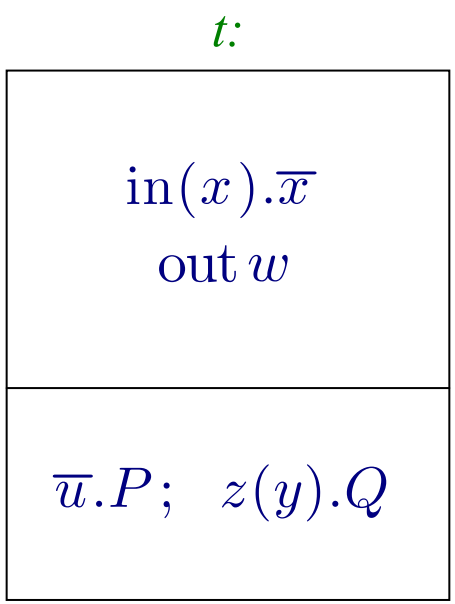
$$
\begin{array}{ll}
(\text{new } tunnel\ @pisa)(& \text{// create a fresh channel, at pisa} \\
\quad \overline{tunnel}\ wischik\ com & \text{// send data 'wischik' over it} \\
\quad \mid tunnel\ (x).\overline{x} & \text{// receive portname, then send on it} \\
) & \\
\mid wischik\ com.\overline{alert} & \text{// when we receive an (empty) msg, alert}
\end{array}
$$

Questions about distribution:

Where is the stuff located on the network?

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

distributed channel machine



channel name of this channel

atoms area, of atoms waiting to rendezvous at this channel

deployment area, of pi terms ready to be executed

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

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

$$(\text{new}\ \ t@p)\big(\overline{t}w \,|\, t(x).\overline{x}\big) \,|\, w.\overline{a}$$

p:
$(\text{new}\, t)\ldots$

w:	*a:*
$\text{in}.\overline{a}$	

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

p:	*(t):*
$\overline{t}w \,|\, t(x).\overline{x}$	

w:	*a:*
$\text{in}.\overline{a}$	

Deploy the input & output atoms
to their appropriate queue

p:	*(t):*
	$\text{out}\, w$ $\text{in}(x).\overline{x}$

w:	*a:*
$\text{in}.\overline{a}$	

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

p:	*(t):*
	$\overline{w}$

w:	*a:*
$\text{in}.\overline{a}$	

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

p:	*(t):*

w:	*a:*
$\text{in}.\overline{a}$ out	

react

p:	*(t):*

w:	*a:*
$\overline{a}$	

deploy

p:	*(t):*

w:	*a:*
	out

... also, garbage-collect (t)

virtual machine, formally

$$u[\mathsf{out}\,x.P\,;\mathsf{in}(y).Q] \xrightarrow{\text{react}} u[P\,;Q\{x/y\}]$$

$$u[\mathsf{out}\,x.P\,;!\mathsf{in}(y).Q] \xrightarrow{\text{react}} u[P\,;Q\{x/y\}\,;!\mathsf{in}(y).Q]$$

$$u[\overline{v}\,x.P]\;\;v[\,] \xrightarrow{\text{dep.out}} u[\,]\;\;v[\mathsf{out}\,x.P]$$

$$u[(\text{new}\;x)P] \xrightarrow{\text{dep.new}} u[P\{x'/x\}]\;\;(\!|x'|\!)[\,] \quad *$$

$$u[P|Q] \xrightarrow{\text{dep.par}} u[P\,;Q]$$

$$u[\mathbf{0}] \xrightarrow{\text{dep.nil}} u[\,]$$

$*$ x' fresh, unique

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

main problem

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

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…

$$(\mathrm{new}\, t)\big(\, u(x).v(y).w(z).\overline{t}xyz \;\mid\; t(xyz).P\big)$$

main problem

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

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

$$P \quad ::= \quad x{=}y \quad | \quad \overline{u}\,\widetilde{x}.P \quad | \quad u\,\widetilde{x}.P \quad | \quad P|P \quad | \quad (x)P \quad | \quad \mathbf{0}$$

$$\overline{u}\,\widetilde{x}.P \,|\, u\,\widetilde{y}.Q \quad \longrightarrow \quad \widetilde{x}{=}\widetilde{y} \,|\, P \,|\, Q$$

$$x{=}y \,|\, P \equiv x{=}y \,|\, P\{y/x\} \qquad \textit{substitution}$$

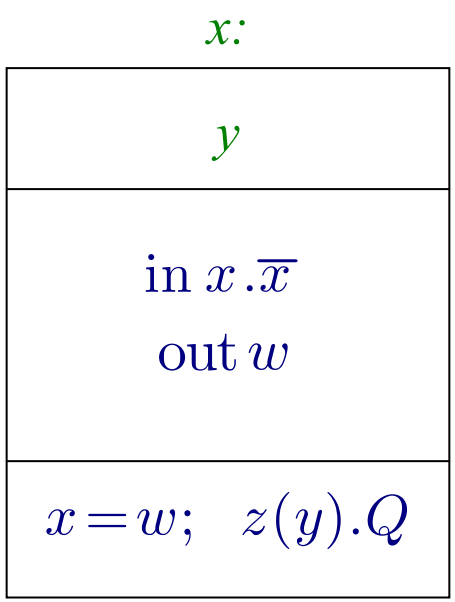
$$(x)(x{=}y) \equiv \mathbf{0} \qquad \textit{local alias}$$

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

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

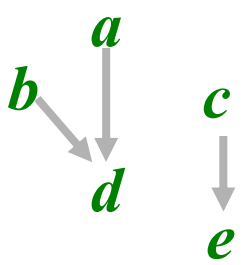
$$x{=}y \,|\, y{=}z \equiv x{=}z \,|\, y{=}z \qquad \textit{transitivity}$$

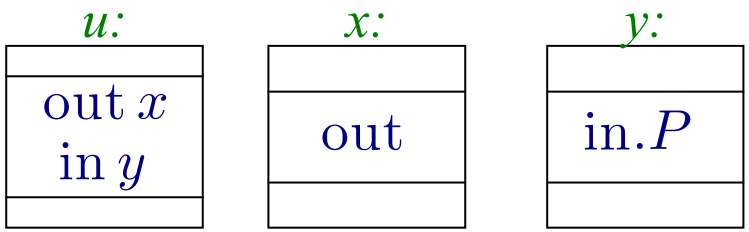
fusion machine



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:

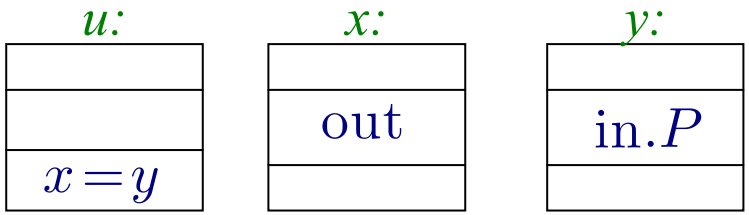




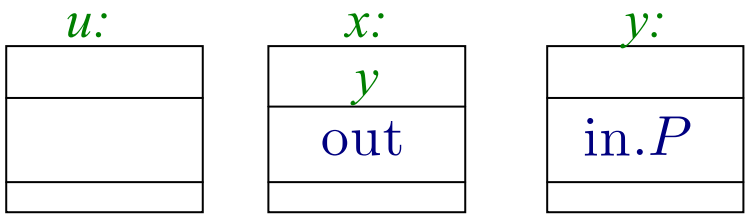
$$\overline{u}x \mid uy \mid \overline{x} \mid y.P \longrightarrow x{=}y \mid \overline{x} \mid y.P$$

$$\longrightarrow x{=}y \mid P$$

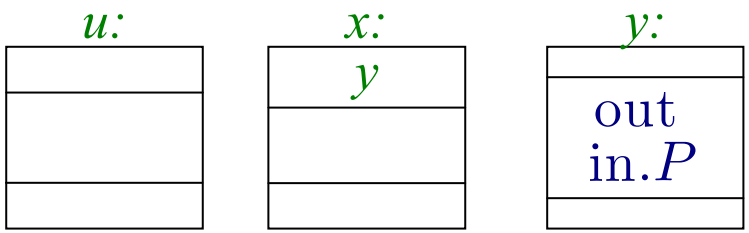
react



deploy fusion by sending to x the message "fuse yourself to y"



migrate atom from x to y



THEOREM

$P \sim Q$ 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:

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

$$
\begin{array}{ll}
(\text{new } xyz, v'@v, w'@w)\ (& \\
\quad ux.\ v'{=}v & // \text{ after } u \text{ has reacted, it tells} \\
\quad \mid v'y.\ w'{=}w & // \ v' \text{ to fuse to } v\text{, so allowing} \\
\quad \mid w'z & // \text{ our } v' \text{ atom to react with } v \text{ atoms} \\
) &
\end{array}
$$

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

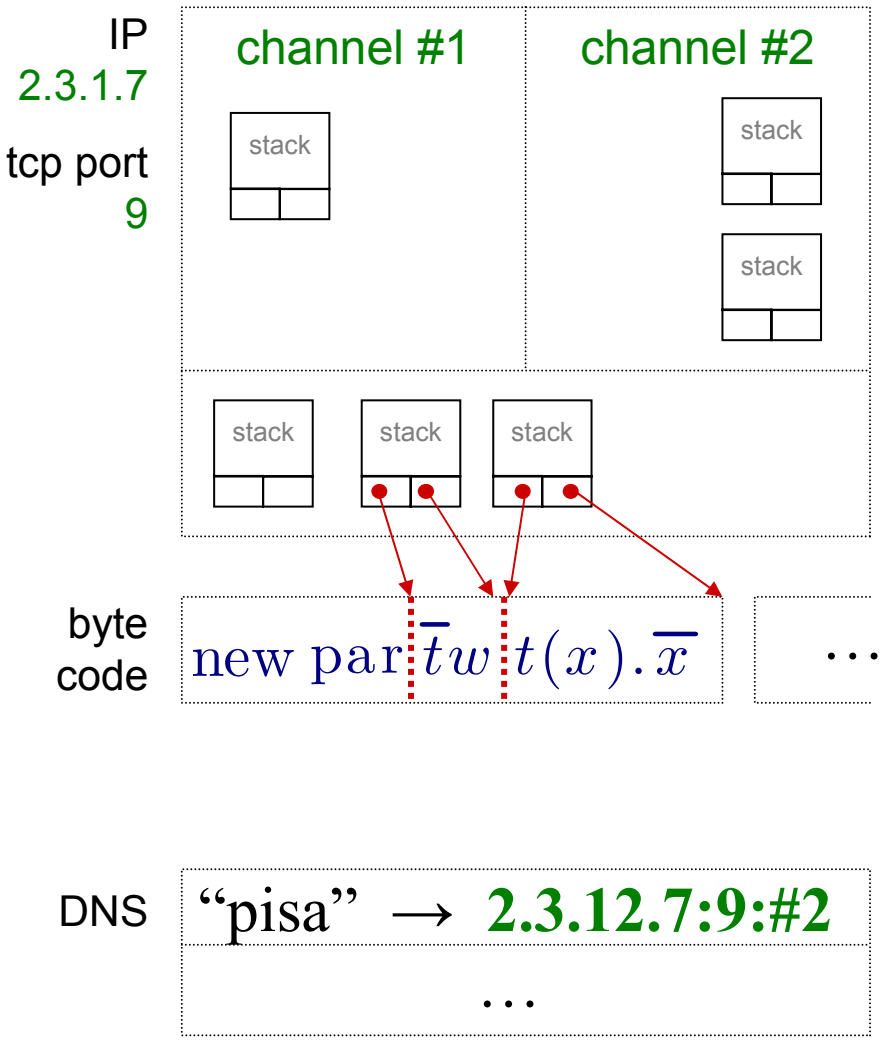
virtual machine, formally

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

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

Processes $P \quad ::= \quad \overline{u}\,\widetilde{x}.P \quad | \quad [!]u(\widetilde{x}).P \quad | \quad (x)P \quad | \quad P|P \quad | \quad \mathbf{0}$

virtual machine in practice



Server thread:

accepts incoming work units over the network

Worker threads:

1. pick up a work unit from the "work bag"
2. if it's PAR, spawn another
3. if it's a remote in/out, send over network
4. if it's a local in/out, either react or add to channel's queue

machine bytecode

Work unit:
(a closure containing a stack, and code pointers)

0:	2.3.1.7 : 9 : 1	
1:	2.3.1.7 : 9 : 2	
2:	14.12.7.5 : 9 : 57	
	code 00	to 110

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

2 *0* *1*

Bytecode:

```
  0  par +80
 10  new @2
 20  par +30
 30  snd 3,0
 40  nil
 50  rcv 3
 60  snd 4
 70  nil
 80  rcv 0
 90  snd 1
100  nil
```

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++

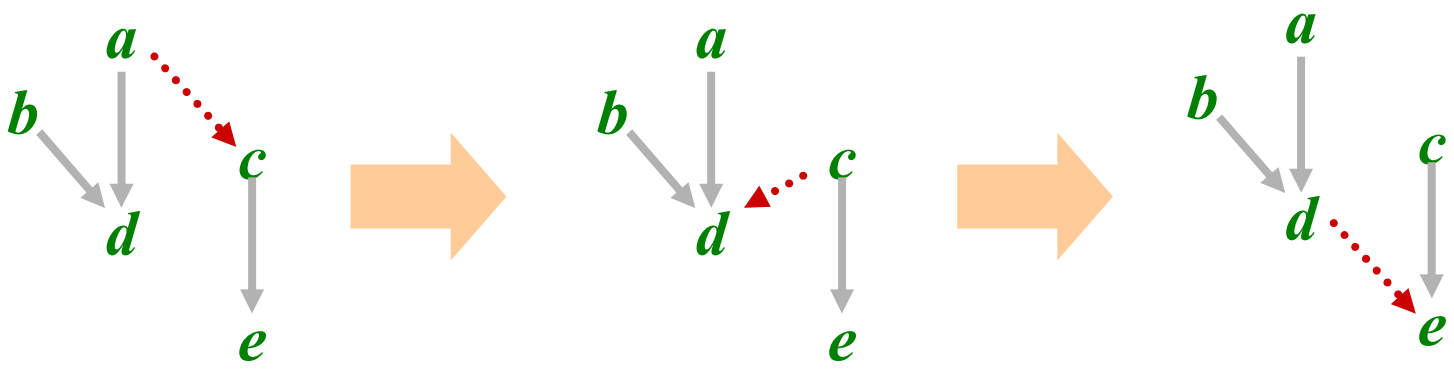
```
{ …                 // there's an implicit continuation K after the rcv,
  rcv(x);           // so we stall the thread and put x.K in the work bag.
  …                 // When K is invoked, it signals the thread to wake up
}
```

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.