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
  wishik.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.

\[
\begin{align*}
&\text{(new } \text{tunnel } @\text{pisa})( \ \\
&\quad \text{tunnel wischik com} \ \\
&\quad \quad \mid \text{tunnel } (x).\overline{x} \\
&\quad \mid \text{wischik com. alert} \\
\end{align*}
\]

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

This one corresponds to  \( t(x).\overline{x} | \overline{t}w | \overline{u}.P | z(y).Q \)
(new \ t@p)(\bar{tw} \ t(x).\bar{x}) \mid w.\bar{a}

**p:**

\[(\text{new } t) \ldots\]

**w:**

\[\text{in.} \bar{a}\]

**a:**

\[
\]

---

**p:**

\[\bar{tw} \ t(x).\bar{x}\]

**w:**

\[\text{in.} \bar{a}\]

**a:**

\[
\]

---

**p:**

\[
\]

**w:**

\[\text{in.} \bar{a}\]

**a:**

\[
\]

---

**p:**

\[\text{out } w\]

**w:**

\[\text{in.} \bar{a}\]

**a:**

\[
\]

---

**p:**

\[
\]

**w:**

\[\text{in.} \bar{a}\]

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

react

deploy

... also, garbage-collect (t)
virtual machine, formally

\[ u[[\text{out } x.P ; \text{in} (y).Q]] \xrightarrow{\text{react}} u[P ; Q\{x/y\}] \]
\[ u[[\text{out } x.P ; !\text{in} (y).Q]] \xrightarrow{\text{react}} u[P ; Q\{x/y\}; !\text{in} (y).Q] \]
\[ u[[v x.P] v[\cdot]] \xrightarrow{\text{dep.out}} u[\cdot] v[[\text{out } x.P]] \]
\[ u[(\text{new } x)P] \xrightarrow{\text{dep.new}} u[P\{x'/x\} \cdot (x')[\cdot]] * \]
\[ u[P|Q] \xrightarrow{\text{dep.par}} u[P ; Q] \]
\[ u[0] \xrightarrow{\text{dep.nil}} u[\cdot] \]

* \(x'\) fresh, unique

**THEOREM** \( P \sim Q \) iff \( u[P] \sim u[Q] \)
Example will transport all of $P$ first to $u$, then $v$, then $w$. How to implement this more efficiently?

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

$$u(x).v(y).w(z).P | \overline{ua} | \overline{vb} | \overline{wc}$$

$$(\text{new } t)(u(x).v(y).w(z).\overline{txyz} | t(xyz).P)$$
Example will transport all of $P$ first to $u$, then $v$, then $w$. How to implement this more efficiently?

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

\[
P ::= \ x = y \ | \ \overline{u} \tilde{x}.P \ | \ u \tilde{x}.P \ | \ P | P \ | \ (x)P \ | \ 0
\]

\[
\overline{u} \tilde{x}.P | u \tilde{y}.Q \rightarrow \tilde{x} = \tilde{y} | P | Q
\]

\[
x = y | P \equiv x = y | P \{y/x\} \quad \text{substitution}
\]

\[
(x)(x = y) \equiv 0 \quad \text{local alias}
\]

\[
x = x \equiv 0 \quad \text{reflexivity}
\]

\[
x = y \equiv y = x \quad \text{symmetry}
\]

\[
x = y | y = z \equiv x = z | y = z \quad \text{transitivity}
\]
fusion machine

\[
x: \\
y \\
in x.\overline{x} \\
out w \\
x = w; \ z(y).Q
\]

**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:
\[ \text{deploy fusion by sending to } x \text{ the message “fuse yourself to } y” \]

\[ \text{migrate atom from } x \text{ to } y \]

\[ \text{THEOREM} \]

\[ P \sim Q \iff u[P] \sim u[Q] \]
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:

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

The code snippet is as follows:

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

$M, M$

$0$

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 $P ::= \overline{u}\tilde{x}.P \mid [!]u(\tilde{x}).P \mid (x)P \mid P|P \mid 0$
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
**Work unit:**
(a closure containing a stack, and code pointers)

<table>
<thead>
<tr>
<th></th>
<th>2.3.1.7 : 9 : 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>2.3.1.7 : 9 : 2</td>
</tr>
<tr>
<td>2:</td>
<td>14.12.7.5 : 9 : 57</td>
</tr>
</tbody>
</table>

```
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

**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 \( \text{snd}(n) \) will invoke the function at that address

Calling \text{snd/rcv} directly from C++

```c
{ ...
    \text{rcv}(x);
    ...
}  // there's an implicit continuation \( K \) after the rcv,
   // 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 \text{pi} code from C++

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

\text{void test2()}
{ ...
}
```
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)
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.

\[
u[x=y] x[p:] \xrightarrow{\text{dep_fu}} u[] x[y:y=p]
\]

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