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

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

This one corresponds to $t(x).\pi | \tau w | \pi.P | z(y).Q$

Questions about distribution:
Where is the stuff located on the network?
How efficiently does it run?
THEOREM \( P \sim Q \) iff \( u[P] \sim u[Q] \)

**main problem**

\[
u(x).v(y).w(z).P \mid \pi a \mid \pi b \mid \pi c
\]

**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…

\[
\text{(new } t \text{) } (u(x).v(y).w(z).Ixyz \mid t(xyz).P)
\]
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...

**fusion machine**

<table>
<thead>
<tr>
<th>x:</th>
<th>y:</th>
</tr>
</thead>
<tbody>
<tr>
<td>in $x$.x</td>
<td>out $w$</td>
</tr>
</tbody>
</table>

**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:

```
        a
      /   \
     b     c
    /     /
   d     e
```

**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:

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

\[
\begin{align*}
\text{(new } x y z, v'@v, u'@u) & \{
ux. v'@v & \quad \text{// after } u \text{ has reacted, it tells} \\
v'y. w'@w & \quad \text{// } v' \text{ to fuse to } v, \text{ so allowing} \\
w'@z & \quad \text{// our } v' \text{ atom to react with } v \text{ atoms}
\}
\end{align*}
\]

Supplemental Slides

- Grammar for fusion machine calculus
- Implementation notes
- Fusion algorithm

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.

Virtual machine, formally

\[
\begin{align*}
\text{Machines } M & ::= \text{u}[^B] & \text{channel machine at } u \\
& | (v)[^B] & \text{private channel machine} \\
& | M \cdot M & \text{0}
\end{align*}
\]

\[
\begin{align*}
\text{Bodies } B & ::= \text{out} \tilde{x}. P & \text{output atom} \\
& | \text{in} (\tilde{x}). P & \text{input atom} \\
& | !\text{in} (\tilde{x}). P & \text{replicated input} \\
& | P & \text{pi process} \\
& | B ; B
\end{align*}
\]

\[
\begin{align*}
\text{Processes } P & ::= \pi \tilde{x}. P & | [v]u[^F]. P & | (x)P & | P|P & | 0
\end{align*}
\]
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)

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PAR +80</td>
</tr>
<tr>
<td>10</td>
<td>new @2</td>
</tr>
<tr>
<td>20</td>
<td>PAR +30</td>
</tr>
<tr>
<td>30</td>
<td>snd 3, 0</td>
</tr>
<tr>
<td>40</td>
<td>nil</td>
</tr>
<tr>
<td>50</td>
<td>rcv 3</td>
</tr>
<tr>
<td>60</td>
<td>snd 4</td>
</tr>
<tr>
<td>70</td>
<td>nil</td>
</tr>
<tr>
<td>80</td>
<td>rcv 0</td>
</tr>
<tr>
<td>90</td>
<td>snd 1</td>
</tr>
<tr>
<td>100</td>
<td>nil</td>
</tr>
</tbody>
</table>

Bytecode:

(fnew (t0<>p)(fW (t(x).π) w.π) 2 2 0 1)

plan: integrate with C++

Treat functions as addresses
- a name = 2.3.1.7 : 9 : 0x04367110
- so that snd(n) will invoke the function at that address

Calling snd/rcv directly from C++
```c
{ ...
  rcv(x);
  // so we stall the thread and put x.K in the work bag
  ...
}
```

Calling arbitrary pi code from C++
```c
pi("u=Ky | Q*"),
pi("w.x.*="fun_as_chan(&test2)*"|Q*"),
void test2()
{ ...
}
```

down: integrate with C++

Calling snd/rcv directly from C++
```c
{ ...
  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 pi code from C++
```c
pi("u=Ky | Q*"),
pi("w.x.*="fun_as_chan(&test2)*"|Q*"),
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.

(fusion merging results)
(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 fulfill this obligation (if $p$ was nil), or will pass it on.