Using ***linear forwarders***
to implement ***input capability***
for implementing the pi calculus

Cosimo Laneve, Philippa Gardner, Lucian Wischik,

Concur, Marseilles September 2003
www.wischik.com/lu/research/linfwd.html

The previous year, at CONCUR, we presented work on our distributed implementation of the pi calculus. It used something called “fusions”. A fusion is something that, when two channels are fused, they can be used interchangeably.

Lots of people found fusions difficult to swallow… also, it seemed hard to make them robust in the presence of failure. So we went back to the drawing board, dissected our implementation, and tried to find out what the core idea inside it was. This talk is about that core idea, “linear forwarders”. We think that linear forwarders make a more simple and elegant story, and one that’s easier to implement.

channel-oriented distribution

$$
\begin{array}{ll}
\text{new } tunnel@mar.(& \text{// Create a fresh channel, at Marseilles} \\
\quad \overline{glob}\,tunnel & \text{// extrude channel } tunnel \\
\quad \mid\ tunnel(svr).svr(c).\bar{c} & \text{// find the current server } svr\text{, ask it what} \\
) & \text{// service } c \text{ it provides, and use the service} \\
\mid\ glob(t).\bar{t}\,server & \text{// Receive } t \text{ and tell it our server} \\
\mid\ \overline{server}\,alert & \text{// (this server allows people to alert)}
\end{array}
$$

Each channel (*tunnel, glob, server, alert*) has a fixed location.

To receive on a channel, must either

- already be at the location (eg. Join), or
- migrate to the location (eg. Fusion Machine), or
- delegate an ***input-proxy*** to that location (eg. Facile, current work)

This example program does none, hence it fails at input on *svr*...

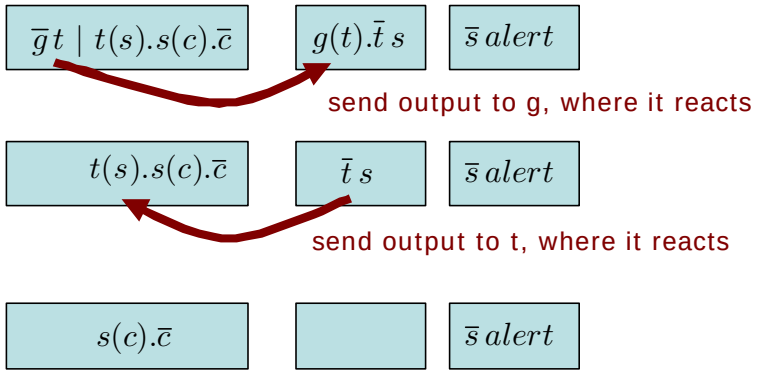
There are two philosophies for implementing concurrent calculi.

(1) Mobile Ambients, Nomadic Pict, this group says that pi-calculus rendezvous is not appropriate for a distributed implementation. These have processes grouped together at locations.

(2) Facile, Join, Fusion machine, Linear Forwarders, MS Biztalk, Highwire – this group says that pi-calculus rendezvous is a good primitive to have. In these, each channel belongs to a single location.

Within the second camp, we face the problem of input capability: as in this example, when a received name is used as the subject of subsequent input.

input capability

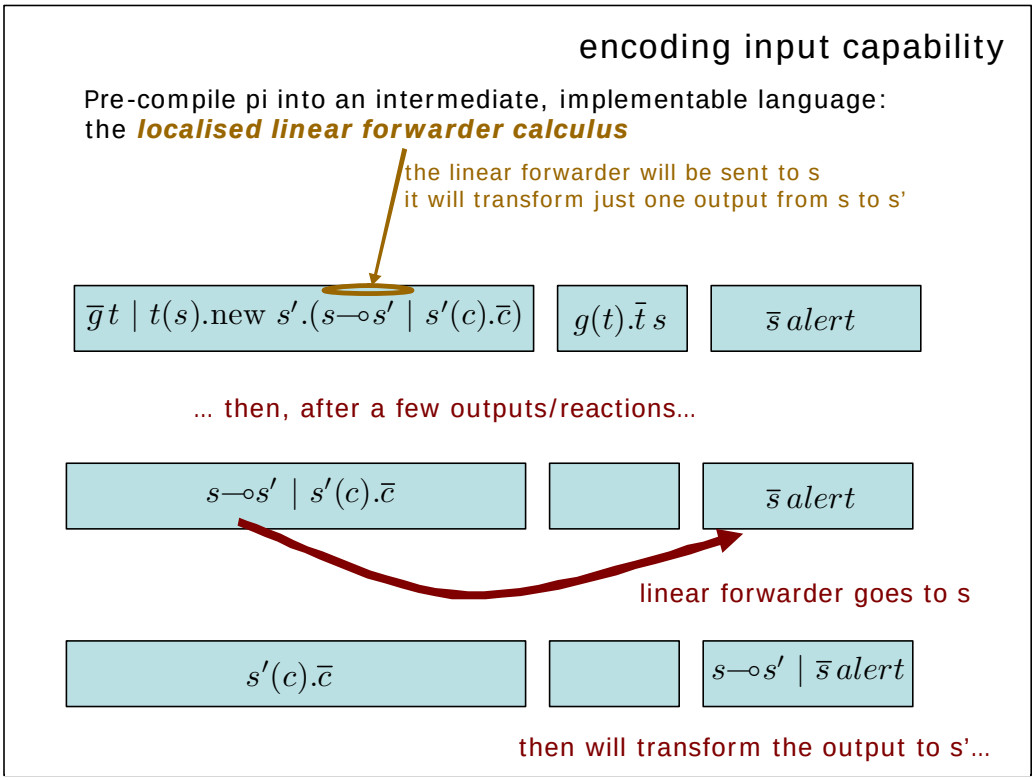


Fails because *s(c)* is at the wrong location to react further.

This is ***input capability***, where a received name *s* is used as the subject of subsequent input.

We'll pre-compile, replacing *input-capability* with *input-proxies*…

This is the step-by-step execution trace of the previous program.



(Step-by-step execution trace after we pre-compile the program into linear forwarders, avoiding input mobility).

Facile was a bit similar. Facile was a synchronous calculus – ie. it had continuations after SEND as well as after RECEIVE. That meant that neither SEND nor RECEIVE could move around in the network. Instead, Facile used output-proxies as well as input-proxies. But it got stuck a bit, because all these proxies made for a difficult protocol that was hard to make robust in the presence of failures.

How do linear forwarders behave in the presence of failure? Well, if a linear forward fails to be delivered to its intended target, that's gives the same results as if it had reached the target and had caused an output to be transformed but then the output was subsequently lost.

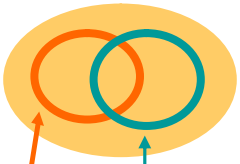
linear forwarder calculus

The ***linear forwarder*** calculus:

$$P ::= \mathbf{0} \;\mid\; \overline{x}\,\widetilde{y} \;\mid\; x(\widetilde{y}) \;\mid\; \nu x.P$$
$$P|P \;\mid\; !P \;\mid\; x \multimap y$$

reaction:

$$\overline{u}\,\widetilde{x} \mid u(\widetilde{y}).P \;\rightarrow\; P\{\widetilde{x}/\widetilde{y}\}$$
$$\overline{u}\,\widetilde{x} \mid u \multimap u' \;\rightarrow\; \overline{u}\,'\widetilde{x}$$



pi subcalculus: subset without linear forwarders $t(s).s(c).\overline{c}$

Localised subcalculus: subset without input capability, ie. *u(x).P* never inputs on *x*

$t(s).\text{new } s'.(s \multimap s' \mid s'(c).\overline{c})$

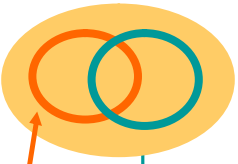
We define the entire linear forwarder calculus first, and then identify two subsets within it.

The linear forwarder calculus is just like the standard pi calculus, but with the addition of a “linear forwarder” term x-oy, and a new reduction rule for it.

linear forwarder calculus

The ***linear forwarder*** calculus:

$$P ::= \mathbf{0} \;\mid\; \overline{x}\,\widetilde{y} \;\mid\; x(\widetilde{y}) \;\mid\; \nu x.P$$
$$P|P \;\mid\; !P \;\mid\; x \multimap y$$



pi subcalculus: subset without linear forwarders $t(s).s(c).\overline{c}$

Localised subcalculus: subset without input capability $t(s).\text{new } s'.(s \multimap s' \mid s'(c).\overline{c})$

We encode **pi** into **Localised**:

$$[\![x(y).P]\!] \;=\; x(y).[\![P]\!]_y$$

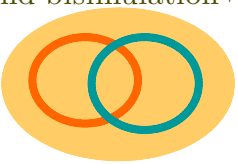
The 'y' subscript means that this name has been received, and so can't be used as subject of subsequent input. Hence, in full...

$$[\![x(\widetilde{y}).P]\!]_{\widetilde{u}} \;=\; \begin{cases} x(\widetilde{y}).[\![P]\!]_{\widetilde{u}\widetilde{y}} & \text{if } x \notin \widetilde{u} \\ \nu x'.(x \multimap x' \mid x'(\widetilde{y}).[\![P]\!]_{\widetilde{u}\widetilde{y}} & \text{if } x \in \widetilde{u} \end{cases}$$

$$[\![\_]\!]_{\widetilde{u}} \;=\; \_$$

As discussed previously, we pre-compile from pi calculus into a localised linear calculus which lacks input mobility.

ground bisimulation $\approx_\ell$



It's all standard, but with one extra rule:

$u(\widetilde{x}).P \xrightarrow{u(\widetilde{x})} P \qquad \overline{u}\,\widetilde{x} \xrightarrow{\overline{u}\,\widetilde{x}} \mathbf{0} \qquad u{\multimap}v \xrightarrow{u(\widetilde{x})} \overline{v}\,\widetilde{x}$

$$\frac{P \xrightarrow{\mu} P' \quad y \notin \mu}{(y)P \xrightarrow{\mu} (y)P'} \qquad \frac{P \xrightarrow{(\widetilde{z})\overline{u}\,\widetilde{x}} P' \quad y \neq u,\ y \in \widetilde{x}\backslash\widetilde{z}}{(y)P \xrightarrow{(y\widetilde{z})\overline{u}\,\widetilde{x}} P'}$$

$$\frac{P|!P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'} \qquad \frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P|Q \xrightarrow{\mu} P'|Q}$$

$$\frac{P \xrightarrow{(\widetilde{z})\overline{u}\,\widetilde{y}} P' \quad Q \xrightarrow{u(\widetilde{x})} Q' \quad \widetilde{z} \cup \mathrm{fn}(Q) = \emptyset}{P|Q \xrightarrow{\tau} (\widetilde{z})(P'|Q'\{\widetilde{y}/\widetilde{x}\})}$$

Let *Ground bisimulation* be the largest symmetric relation $\approx_\ell$ such that $P \approx_\ell Q$ and $P \xrightarrow{\mu} P'$ implies $Q \stackrel{\mu}{\Longrightarrow} \approx_\ell Q'$.
Common result: in asynchronous calculi, $\approx_\ell$ is a congruence.

The labelled transition system for the linear forwarder calculus.

- Pi contexts are exactly as discriminating as (localised) linear-forwarder contexts.

  Or, by limiting general input capability to only input capability of linear forwarders, we keep the same expressive power of terms and the same distinguishing power of contexts.

  $P \approx_\pi Q \quad \text{iff} \quad [\![P]\!] \approx [\![Q]\!]$

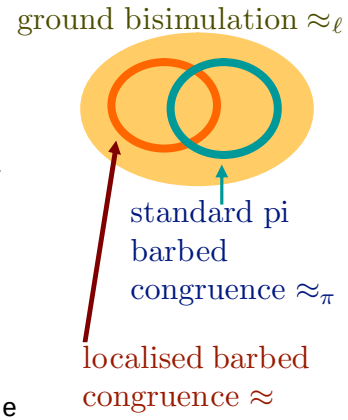
- Ground bisimulation helps as a proof technique

  $P \approx_\ell [\![P]\!] \approx_\ell [\![P]\!]_{\tilde{u}} \qquad \approx \supset \approx_\ell|_{\text{localised}}$

  but it's too strict in general, eg. for this standard result from equators:

  $u(x).P \approx u(x').\nu x.(!x{\multimap}x' \mid !x'{\multimap}x \mid P)$

  $\not\approx_\ell$



The first result, "full abstraction", is what justifies our work.

We really have three separate equivalence relations that we're juggling with. The ground bisimulation from the previous slide is helpful in proofs – it's as though it were tailor-made specially to prove our encoding correct! But it's too strict for general use. So instead we use barbed congruence.

A "congruence" is a relation that's closed under contexts. Obviously you have to specify what those contexts should be. The standard barbed congruence is closed under pi-contexts. Our localised barbed congruence is closed under localised-contexts. We might also have defined a "linear forwarder congruence" that was closed under all linear forwarder contexts, but there didn't seem much use. In any case, thanks to the ground-bisimulation results about the encoding, it'd be just the same.

It's worth being careful about the statement of ground bisimulation. "If two terms are in the localised subcalculus, and they are judged equivalent according to ground bisimulation, then they are also judged equivalent by localised-barbed-congruence".

linearity of forwarders

$$\overline{u}\,x \mid u(y).y().P$$

$$!x{\multimap}y \mid y().P$$

$$\nu y'.(x{\multimap}y' \mid y'().P)$$

A ***non-linear*** use of forwarders:

Reaction gives a substitution *{x/y}*. This can be implemented by forwarding all future *x*s to *y*.

(similar to equator work by Mero, pi-I work by Boreale, our earlier work on fusions)

Our ***linear*** use of forwarders:

Reaction gives a substitution *{x/y}*. In our subsequent source code we have only one input on *x*, so we need forward only one *x* to our input.

Non-linearity here would be an error:

$$\nu y'.(!x{\multimap}y' \mid y'().P)$$

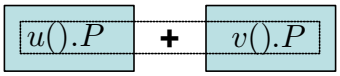
even after our input y'() had been used up, it'd keep forwarding to y', so denying other inputs their chance.

The interesting thing is about linearity. Really, we have given a calculus that uses forwarders for the code P that's inside a term. There's a finite amount of code, and so we have LINEAR forwarders.

Other people have used forwarders for the possible data that might be changed subsequently. There's a potentially infinite amount of external data that might have to be changed, so these people have to use NON-LINEAR forwarders.

(Actually, the equation given here doesn't quite work as it is. It needs either a non-linear forwarder in the reverse direction, giving equators, or it needs some kind of protocol around it as used by Boreale, or it needs a more complicated mechanism as we did with fusions.)

distributed choice



Distributed choice is hard: any proposed reaction at *u* would have to ask permission from *v* before proceeding.
"Where is the + implemented?"

So we encode it with linear forwarders:
forward *u* and *v* both to a common location.
(and after a choice has been made, ***undo*** the other forwarder)

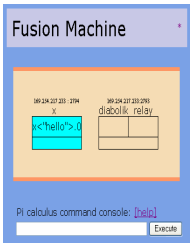
$$[\![u().P + v().Q]\!] = \nu u'v'.\big(u{\multimap}u' \mid v{\multimap}v' \;\mid\; u'().(P|v'{\multimap}v) + v'().(Q|u'{\multimap}u)\big)$$

the fusion project at bologna

www.cs.unibo.it/fusion

Cosimo Laneve
Lucian Wischik
Manuel Mazzara
Fabrizio Bisi
Paolo Milazzo

prototype implementation in Java



distributed implementation in C++/Win32

```
spawn {
  channel K =
     new channel<Result>;
  mailer.send(email,
     sub,body,response);
  K.recv(r: Result);
  if (r!="ok")
     pending.send(email);
}
```

Java impl. with XML data-types

The first two implementations, by Wischik, used fusions. The third implementation, by Paolo and Fabrizio, uses linear forwarders.