# Linear Forwarders

Philippa Gardner[1], Cosimo Laneve[2], and Lucian Wischik[2]

[1] Imperial College, London. pg@doc.ic.ac.uk
[2] University of Bologna, Italy. laneve@cs.unibo.it, lu@wischik.com

**Abstract.** A *linear forwarder* is a process which receives one message on a channel and sends it on a different channel. Such a process allows for a simple implementation of the asynchronous pi calculus, by means of a direct encoding of the pi calculus' *input capability* (i.e. where a received name is used as the subject of subsequent input). This encoding is fully abstract with respect to barbed congruence.
Linear forwarders are actually the basic mechanism of an earlier implementation of the pi calculus, the *fusion machine*. We modify the machine, replacing fusions by forwarders. The result is more robust in the presence of failures, and more fundamental.

## 1 Introduction

In the pi calculus, a program has a collection of channels, and it runs through interaction over these channels. A possible distributed implementation is to let each channel belong to a single location. For instance, there is one location for the channels $u, v, w$ and another for $x, y, z$, and the input resource $u(a).P$ goes in the first location. If an output $\overline{u}\,x$ should arise anywhere else in the system, it knows where it can find a matching input. This basic scheme is used in the join calculus [**?**], in the $\pi_{1\ell}$ calculus [**?**], and in the fusion machine [**?**]. (A different approach is taken in D$\pi$ [**?**], in nomadic pict [**?**], and in the ambient calculus [**?**], where agent migration commands are used for remote interaction.)

We immediately face the problem of **input capability**, which is the ability in the pi calculus to receive a channel name and subsequently accept input on it. Consider the example $x(u).u(v).Q$. This program is located at (the location of) $x$, but upon reaction with $\overline{x}\,w$ it produces the continuation $w(v).Q\{^w/_u\}$ — and this continuation is still at $x$, whereas it should actually be at $w$. Solving the problem of input capability is the key challenge in a distributed implementation of the pi calculus.

The point of this paper is to solve the problem of input capability with a language that is "just right" – it neither disallows more features than necessary (as does the join calculus), nor adds more implementation work than is necessary (as does the fusion machine). One measure of our solution is that we obtain full abstraction with the pi calculus, with respect to weak barbed congruence. (Here and henceforth, we refer implicitly to the *asynchronous* pi calculus).

First of all, let us consider in more detail the other solutions to input capability. The join calculus and localised pi calculus [**?**] simply disallow it: that is, in

a term $x(u).P$, then $P$ may not contain any inputs on channel $u$. The problem now is how to encode input capability into such a *localised* calculus. Such an encoding is possible, but awkward: when the term $x(u).u(v).Q \mid \overline{x}\, w$ is encoded and then performs the reaction, it does not perform the substitution $\{^w/_u\}$, but rather encodes this substitution as a persistent forwarder between $w$ and $u$. Next, a firewall is needed to protect the protocol used by these forwarders. (The forwarder is called a "merged proxy pair" in the join calculus).

The fusion machine instead implements input capability through the runtime migration of code. In our example, $w(v).Q\{^w/_u\}$ would migrate from $x$ over to $w$ after the interaction. The migration is costly, however, when the continuation $Q$ is large; and code migration requires an elaborate infrastructure. To mitigate this, a large amount of the work on the fusion machine involved an encoding of arbitrary programs into *solos* programs (ones which have only simple continuations) without incurring a performance penalty. But the encoding used fusions, implemented through persistent trees of forwarders, which seem awkward and fragile in the presence of failures.

The solution presented in this paper is to disallow general input capability, and to introduce instead a limited form of input, the **linear forwarder**. A linear forwarder $x \multimap y$ is a process which allows just one $x$ to be turned into a $y$. Importantly, this limited form can be used to easily encode general input capability. For example, consider the pi calculus term $x(u).u(v).Q$. We will encode it as

$$x(u).(u')(u \multimap u' \mid u'(v).Q')$$

where the input $u(v)$ has been turned into a local input $u'(v)$ (at the same location as $x$), and where the forwarder allows one output on $u$ to interact with $u'$ instead. The encoding has the property that if the forwarder $u \multimap u'$ exists, then there is guaranteed to be an available input on $u'$. We remark that linearity is crucial: if the forwarder persisted, then the guarantee would be broken; any further $u$ turned into $u'$ would become inert since there are no other inputs on $u'$.

One might think of a linear forwarder $x \multimap y$ as the pi calculus agent $x(\widetilde{u}).\overline{y}\,\widetilde{u}$ located at $x$. This agent would be suitable for a point-to-point network such as the Internet. But we have actually turned forwarders into first-class operators in order to abstract away from any particular implementation. This is because other kinds of networks benefit from different implementations of linear forwarders. In a broadcast network, $x \multimap y$ might be located at $y$; whenever it hears an offer of $\overline{x}\,\widetilde{u}$ being broadcast, the machine at $y$ can take up the offer. Another possibility is to use a shared tuple-space such as Linda [**?**], and ignore all linearity information. (The fusion machine also amounts to a shared state which ignores linearity).

In this paper we show how to encode the pi calculus into a linear forwarder calculus; conversely, we also show how linear forwarders can be encoded into the pi calculus. We therefore obtain full abstraction with respect to barbed congruence.

We also describe a *linear forwarder machine*. It is a simplified form of our earlier fusion machine, and more robust with respect to failures. This machine

gives an implementation of distributed rendezvous which can be performed locally. In this respect it is different from Facile [**?**], which assumes a three-party handshake. (This handshake is a protocol for interaction, and so prevents full abstraction.) We prove full abstraction between the machine and the linear forwarder calculus, with respect to barbed congruence.

*Related work.* Forwarders have already been studied in detail by the pi community. Much work centres around the $\pi$I calculus [**?**] – a variant of the pi calculus in which only private names may be emitted, as in $(x)\overline{u}\,x$. Boreale uses forwarders to encode the emission of free names [**?**]: the reaction $u(a).Q \mid \overline{u}\,x$ does not perform the substitution $\{x/a\}$, but instead encodes it as a persistent forwarder between $a$ and $x$. The same technique is used by Merro and Sangiorgi [**?**] in proofs about the localised pi calculus; and both are inspired by Honda's *equators* [**?**], which are bidirectional forwarders. Something similar is also used by Abadi and Fournet [**?**]. When channels are used linearly, Kobayashi et al. [**?**] show that a linear forwarder can simulate a substitution.

We remark upon some differences. If substitutions are encoded as persistent forwarders, then the ongoing execution of a program will create steadily more forwarders. In contrast, we perform substitution directly, and in our setting the number of forwarders decreases with execution. More fundamentally, the $\pi$I calculus uses forwarders to effect the substitution of data, and they must be persistent (nonlinear) since the data might be used arbitrarily many times by contexts. We use forwarders to effect the input capability of code, and this is linear because a given piece of source code contains only finitely many input commands. Our proofs actually have the same structure as those of Boreale, but are much simpler due to linearity.

*Structure.* The structure of this paper is as follows. Section 2 gives the linear forwarder calculus, and shows how to encode the pi calculus (with its input mobility) into this calculus. Section 3 gives bisimulations for the linear forwarder calculus, and Section 4 proves full abstraction of the pi calculus encoding. Section 5 describes a distributed abstract machine for implementing the linear forwarder calculus, and Section 6 proves full abstraction for this implementation. We outline future developments in Section 7.

## 2 The linear forwarder calculus

We assume an infinite set $\mathcal{N}$ of *names* ranged over by $u, v, x, \ldots$. Names represent communication channels, which are also the values being transmitted in communications. We write $\widetilde{x}$ for a (possibly empty) finite sequence $x_1 \cdots x_n$ of names. *Name substitutions* $\{\widetilde{y}/\widetilde{x}\}$ are as usual.

**Definition 1 (Linear forwarder calculus)** *Terms are given by*

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

*Structural congruence $\equiv$ is the smallest equivalence relation satisfying the following and closed with respect to contexts and alpha-renaming:*

$$P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad !P \equiv P|!P$$
$$(x)(y)P \equiv (y)(x)P \qquad (x)(P|Q) \equiv P \mid (x)Q \quad \textit{if } x \notin \mathrm{fn}\, P$$

*Reaction is the smallest relation satisfying the following axioms and closed under $\equiv$, $(x)\_$ and $\_ \mid \_$:*

$$\begin{aligned} u(\widetilde{x}).P \mid \overline{u}\,\widetilde{y} &\;\rightarrow\; P\{\widetilde{y}/\widetilde{x}\} \\ \overline{x}\,\widetilde{u} \mid x{\multimap}y &\;\rightarrow\; \overline{y}\,\widetilde{u} \end{aligned}$$

The operators in the syntax are all standard apart from the linear forwarder $x{\multimap}y$. This allows one output on $x$ to be transformed into one on $y$, through the second reaction rule.

In the output $\overline{x}\,\widetilde{y}$ and the input $x(\widetilde{y}).P$, the name $x$ is called the *subject* and the names $\widetilde{y}$ are the *objects*. In $(x)P$, the name $x$ is said to be *bound*. Similarly, in $x(\widetilde{y}).P$, the names $\widetilde{y}$ are bound in $P$. The *free names* in $P$, denoted $\mathrm{fn}(P)$, are the names in $P$ with a non-bound occurrence. We write $(x_1 \cdots x_n)P$ for $(x_1) \cdots (x_n)P$.

Next we make a *localised* sub-calculus, by adding the *no-input-capability* constraint. It is standard from the $\pi_L$ calculus [**?**] and the join calculus that such a constraint makes a calculus amenable to distributed implementation.

**Definition 2 (Localised calculus)** *The* localised *linear forwarder calculus, which we abbreviate $L\ell$, is the sub-calculus of the linear forwarder calculus which satisfies the* no-input-capability *constraint: in $x(\widetilde{u}).P$, $P$ has no free occurrence of $\widetilde{u}$ as the subject of an input.*

We remark that the no-input-capability constraint is preserved by structural congruence and by reaction.

A well known sub-calculus of the linear forwarder calculus — the asynchronous pi calculus [**?**] — is obtained by dropping linear forwarders. The pi calculus' input capability may be encoded into the localised calculus as follows.

**Definition 3 (Encoding pi)** *The encoding $[\![\cdot]\!]$ maps terms in the pi calculus into terms in the $L\ell$ calculus as follows. (In the input and restriction cases, assume that the bound names do not clash with $\widetilde{u}$.) Define $[\![P]\!] = [\![P]\!]_{\emptyset}$.*

$$\begin{aligned}
[\![x(\widetilde{y}).P]\!]_{\widetilde{u}} &= \begin{cases} x(\widetilde{y}).[\![P]\!]_{\widetilde{u}\widetilde{y}} & \textit{if } x \notin \widetilde{u} \\ (u_i')(u_i{\multimap}u_i' \mid u_i'(\widetilde{y}).[\![P]\!]_{\widetilde{u}\widetilde{y}}) & \textit{if } x = u_i, u_i \in \widetilde{u} \end{cases} \\
[\![(x)P]\!]_{\widetilde{u}} &= (x)([\![P]\!]_{\widetilde{u}}) \\
[\![P|Q]\!]_{\widetilde{u}} &= [\![P]\!]_{\widetilde{u}} \mid [\![Q]\!]_{\widetilde{u}} \\
[\![!P]\!]_{\widetilde{u}} &= ![\![P]\!]_{\widetilde{u}} \\
[\![\overline{x}\,\widetilde{y}]\!]_{\widetilde{u}} &= \overline{x}\,\widetilde{y} \\
[\![\mathbf{0}]\!]_{\widetilde{u}} &= \mathbf{0}
\end{aligned}$$

To understand the encoding, note that we use "primed" names to denote local copies of names. So the encoding of $x(u).u(y).P$ will use a new channel $u'$ and a process $u'(y).P$, both at the same location as $x$. It will also create exactly one forwarder $u \multimap u'$, from the argument passed at runtime to $u'$. Meanwhile, any output use of $u$ is left unchanged.

To illustrate the connection between the reactions of a term and of its translation, we consider the pi calculus reduction $\overline{u}\,y \mid u(x).P \;\to\; P\{y/x\}$. By translating we obtain:

$$\begin{array}{rcl} [\![\, \overline{u}\,y \mid u(x).P \,]\!]_u & = & \overline{u}\,y \mid (u')(u \multimap u' \mid u'(x).[\![P]\!]_{xu}) \\ & \to & (u')(\overline{u}\,'y \mid u'(x).[\![P]\!]_{xu}) \\ & \to & (u')([\![P]\!]_{xu}\{y/x\}) \\ & \equiv & [\![P]\!]_{xu}\{y/x\} \end{array}$$

Note that the final state of the translated term is subscripted on $x$ and $u$, not just on $u$. In effect, the translated term ends up with some garbage that was not present in the original. Because of this garbage, it is not in general true that $Q \to Q'$ implies $[\![Q]\!] \to^* [\![Q']\!]$; instead we must work up to some behavioural congruence. The following section deals with barbed congruence.

We remark that linearity is crucial in the translation. For instance, consider a nonlinear translation where forwarders are replicated:

$$[\![u(x).P]\!]_u \;=\; (u')(!u \multimap u' \mid u'(y).P)$$

Then consider the example

$$\begin{array}{rcl} [\![u().P \mid u().Q \mid \overline{u} \mid \overline{u}]\!]_u & = & (u')(!u \multimap u' \mid u'().P) \mid (u'')(!u \multimap u'' \mid u''().Q) \mid \overline{u} \mid \overline{u} \\ & \to & (u')(P \mid !u \multimap u') \mid (u'')(!u \multimap u'' \mid u''().Q) \mid \overline{u} \\ & \to & (u')(P \mid \overline{u}' \mid !u \multimap u') \mid (u'')(!u \multimap u'' \mid u''().Q) \end{array}$$

Here, both outputs were forwarded to the local name $u'$, even though the resource $u'().P$ had already been used up by the first one. This precludes the second one from reacting with $Q$ — a reaction that would have been possible in the original pi calculus term. We need linearity to prevent the possibility of such dead ends.

## 3 Bisimulation and congruence

We use barbed congruence [**?**] as our semantics for the $L\ell$ calculus.

**Definition 4 (Barbed congruence)** *The observation relation $P \downarrow u$ is the smallest relation generated by*

$$\overline{u}\,\widetilde{x} \downarrow u \qquad\qquad P|Q \downarrow u \quad \text{if } P \downarrow u \text{ or } Q \downarrow u$$
$$(x)P \downarrow u \quad \text{if } P \downarrow u \text{ and } u \neq x \qquad\qquad !P \downarrow u \quad \text{if } P \downarrow u$$

*We write $\Downarrow$ for $\to^*\downarrow$ and $\Rightarrow$ for $\to^*$. A symmetric relation $\mathcal{R}$ is a weak barbed* bisimulation *if whenever $P\,\mathcal{R}\,Q$ then*

1. $P \Downarrow u$ *implies* $Q \Downarrow u$
2. $P \to P'$ *implies* $Q \Rightarrow Q'$ *such that* $P' \mathcal{R} Q'$

*Let* $\dot{\approx}$ *be the largest weak barbed bisimulation. Two terms* $P$ *and* $Q$ *are weak barbed* congruent *in the* $L\ell$ *calculus, when for every* $C$*, then* $C[P] \dot{\approx} C[Q]$*, where* $C[P]$ *and* $C[Q]$ *are assumed to be terms in the* $L\ell$ *calculus. Let* $\approx$ *be the least relation that relates all congruent terms.*

We remark that barbed bisimulation $\dot{\approx}$ is defined for the linear forwarder calculus. However, the weak barbed *congruence* $\approx$ is predicated upon the $L\ell$ subcalculus. Similar definitions may be given for the pi calculus, and, with abuse of notation, we keep $\dot{\approx}$ and $\approx$ denoting the corresponding semantics.

As an example of $\approx$ congruent terms in the $L\ell$ calculus, we remark that

$$u(x).P \quad \approx \quad u(x').(x)(!x{-\!\circ}x' \mid !x'{-\!\circ}x \mid P). \tag{1}$$

This is a straightforward variant of a standard result to do with equators [**?**], and we use it in Lemma 9.

Our overall goal is to prove that the encoding $[\![\cdot]\!]$ preserves the $\approx$ congruence. The issue, as described near the end of the previous section, is that an encoded term may leave behind garbage. To show that it is indeed garbage, we must prove that $[\![P]\!]_u$ and $[\![P]\!]_{ux}$ are congruent. But the barbed semantics offer too weak an induction hypothesis for this proof. A standard alternative technique (used eg. by Boreale [**?**]) is to use barbed semantics as the primary definition, but then to use in the proofs a labelled transition semantics and its corresponding bisimulation — which is stronger than barbed congruence. The remainder of this section is devoted to the labelled semantics.

**Definition 5 (Labelled semantics)** *The labels, ranged over by* $\mu$*, are the standard labels for interaction* $\xrightarrow{\tau}$*, input* $\xrightarrow{u(\widetilde{x})}$ *and possibly-bound output* $\xrightarrow{(\widetilde{z})\overline{u}\,\widetilde{x}}$ *where* $\widetilde{z} \subseteq \widetilde{x}$*. The bound names* $\mathrm{bn}(\mu)$ *of these input and output labels are* $\widetilde{x}$ *and* $\widetilde{z}$ *respectively.*

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

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

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

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

*The transitions of* $P|Q$ *have mirror cases, which we have omitted. We implicitly identify terms up to alpha-renaming* $\equiv_\alpha$*: that is, if* $P \equiv_\alpha \xrightarrow{\mu} P'$ *then* $P \xrightarrow{\mu} P'$*. We write* $\stackrel{\mu}{\Longrightarrow}$ *for* $\xrightarrow{\tau}{}^{*}\xrightarrow{\mu}$*.*

*A symmetric relation $\mathcal{R}$ is a weak labelled bisimulation if whenever $P\ \mathcal{R}\ Q$ then $P \xrightarrow{\mu} P'$ implies $Q \stackrel{\mu}{\Longrightarrow} Q'$. Let $\approx_\ell$ be the largest labelled bisimulation.*

It is a standard result that $\approx_\ell$ is a congruence with respect to contexts in the full linear forwarder calculus (and hence also with respect to contexts in the $L\ell$ calculus and the pi calculus). The connection between labelled and barbed semantics is also standard:

**Lemma 6** *In the $L\ell$ calculus,*

1. $P \to P'$ *iff* $P \xrightarrow{\tau} \equiv P'$
2. $P \downarrow u$ *iff* $P \xrightarrow{(\widetilde{z})\overline{u}\,\widetilde{x}} P'$
3. $\approx_\ell \subset \approx$.

The bisimulation $\approx_\ell$ allows for some congruence properties to be proved trivially: (the first will be used in Proposition 10)

$$\begin{array}{c} u{-\!\circ}v \ \approx_\ell\ u(\widetilde{x}).\overline{v}\,\widetilde{x} \ \approx_\ell\ (u')(u{-\!\circ}u' \mid u'(\widetilde{x}).\overline{v}\,\widetilde{x}). \\ u(x).P \ \approx_\ell\ (u')(u{-\!\circ}u' \mid u'(x).P). \end{array} \tag{2}$$

## 4 Full abstraction for the pi calculus encoding

The $L\ell$ calculus is fully abstract with respect to the pi calculus encoding: $P \approx Q$ in pi if and only if $[\![P]\!] \approx [\![Q]\!]$ in $L\ell$ . Informally, this is because the pi calculus input capability can be encoded with linear forwarders (as in Definition 3); and conversely a linear forwarder $x{-\!\circ}y$ can be encoded as the pi calculus term $x(\widetilde{u}).\overline{y}\,\widetilde{u}$. This section builds up to a formal proof of the result.

The structure of the proof follows that of Boreale ([**?**], Definition 2.5 to Proposition 3.6). However, the proofs are significantly easier in our setting. We begin with a basic lemma about the encoding $[\![\cdot]\!]$.

**Lemma 7**

1. $[\![P]\!]_{\widetilde{z}} \approx_\ell [\![P]\!]_{x\widetilde{z}}$.
2. $[\![P]\!]_{\widetilde{z}}\{y/x\} \approx_\ell [\![P\{y/x\}]\!]_{\widetilde{z}}$.
3. $[\![P]\!]_{\widetilde{z}}\{\widetilde{y}/\widetilde{x}\} \approx_\ell (u')(\overline{u}\,'\widetilde{y} \mid u'(\widetilde{x}).[\![P]\!]_{\widetilde{z}\widetilde{x}})$.

*Proof.* The first two are trivial inductions on $P$. The last one follows directly. □

We draw attention to the first part of Lemma 7. This is an important simplifying tool. It means that, even though the encoding $[\![u(x).P]\!]_u = (u')(u'{-\!\circ}u \mid u'(x).[\![P]\!]_{ux})$ involves progressively more subscripts, they can be ignored up to behavioural equivalence. Thus, although a context $C$ might receive names $\widetilde{x}$ in input, we can ignore this fact: $C\big[[\![P]\!]_{\widetilde{x}}\big] \approx_\ell C\big[[\![P]\!]\big]$. Part 1 does not hold for Boreale, and so his equivalent of Part 3 uses a significantly longer (5-page) alternative proof. In the Appendix we give a more detailed comparison with Boreale's work, and also remark upon simpler but unconventional proofs for this section.

The following proposition is a preliminary step towards full abstraction. We defer the proof to the Appendix, since it is similar to Boreale's proof.

**Proposition 8** *For $P, Q$ in the pi calculus, $P \mathrel{\dot{\approx}} Q$ if and only if $[\![P]\!] \mathrel{\dot{\approx}} [\![Q]\!]$.*

We will also need the following lemma. It generalises Lemma 7.1 to apply to barbed congruence rather than just labelled bisimulation. Effectively, it implies that a non-localised context can be transformed into a localised one.

**Lemma 9** *For $P, Q$ in the pi calculus, $[\![P]\!] \approx [\![Q]\!]$ implies $[\![P]\!]_{\widetilde{z}} \approx [\![Q]\!]_{\widetilde{z}}$.*

*Proof.* From Lemma 7.1 we get $[\![P]\!]_{\widetilde{z}} \approx_\ell [\![P]\!] \approx [\![Q]\!] \approx_\ell [\![Q]\!]_{\widetilde{z}}$. The result follows by Lemma 6.3 and the transitivity of $\approx$. (We thank an anonymous reviewer for this proof.) □

We are now ready to establish full abstraction for the encoding of the pi calculus into the $L\ell$ calculus.

**Theorem 10 (Full abstraction)** *For $P, Q$ in the pi calculus, $P \approx Q$ if and only if $[\![P]\!] \approx [\![Q]\!]$ in the $L\ell$ calculus.*

*Proof.* We demonstrate that (1) $P \not\approx Q$ implies $[\![P]\!] \not\approx [\![Q]\!]$ and, conversely, (2) $[\![P]\!] \not\approx [\![Q]\!]$ implies $P \not\approx Q$. We write $C_\pi$ and $C_{L\ell}$ to range over contexts such that $C_\pi[P], C_\pi[Q]$ are terms in the pi calculus, and $C_{L\ell}\big[[\![P]\!]\big], C_{L\ell}\big[[\![Q]\!]\big]$ are terms in the $L\ell$ calculus.

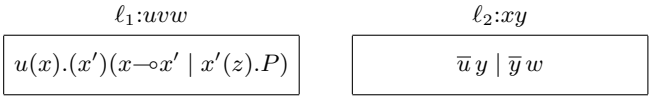
To establish (1), extend the translation $[\![\cdot]\!]$ to contexts in the obvious way. Since the translation $[\![\cdot]\!]$ is compositional, we get $[\![C[P]]\!] = [\![C]\!]\big[[\![P]\!]_{\widetilde{z}}\big]$ and $[\![C[Q]]\!] = [\![C]\!]\big[[\![Q]\!]_{\widetilde{z}}\big]$ for some $\widetilde{z}$ determined by $C$. Next, we reason by contradiction: we prove that $P \not\approx Q$ and $[\![P]\!] \approx [\![Q]\!]$ is false. Since $P \not\approx Q$, there exists a context $C_\pi[\,]$ such that $C_\pi[P] \mathrel{\dot{\not\approx}} C_\pi[Q]$. By $[\![P]\!] \approx [\![Q]\!]$ and Lemma 9, we also have $[\![P]\!]_{\widetilde{z}} \approx [\![Q]\!]_{\widetilde{z}}$. Therefore, in particular $[\![C_\pi]\!]\big[[\![P]\!]_{\widetilde{z}}\big] \mathrel{\dot{\approx}} [\![C_\pi]\!]\big[[\![Q]\!]_{\widetilde{z}}\big]$ and, by the above equalities, $[\![C_\pi[P]]\!] \mathrel{\dot{\approx}} [\![C_\pi[Q]]\!]$. By Proposition 8, this latter bisimulation contradicts $C_\pi[P] \mathrel{\dot{\not\approx}} C_\pi[Q]$.

To establish (2), we show that pi contexts are as expressive as linear forwarder contexts, by exhibiting a pi implementation of linear forwarders. To this end, we define $\widehat{\cdot}$, which translates $x{-\!\circ}y$ into $x(\widetilde{u}).\overline{y}\,\widetilde{u}$ and leaves all else unchanged. Similarly to (1), we prove that $[\![P]\!] \not\approx [\![Q]\!]$ and $P \approx Q$ are contradictory. We are given a context $C_{L\ell}[\,]$ such that $C_{L\ell}[[\![P]\!]] \mathrel{\dot{\not\approx}} C_{L\ell}[[\![Q]\!]]$. Consider the agent $[\![\widehat{C_{L\ell}}[P]]\!]$, which by definition is equal to $[\![\widehat{C_{L\ell}}]\!]\big[[\![P]\!]_{\widetilde{z}}\big]$ for some $\widetilde{z}$. By Lemma 7 this is $\approx_\ell$ to $[\![\widehat{C_{L\ell}}]\!]\big[[\![P]\!]\big]$. Now we consider the double translation $[\![\widehat{\cdot}]\!]$; it will convert each forwarder $u{-\!\circ}v$ into either $u(\widetilde{x}).\overline{v}\,\widetilde{x}$ or $(u')(u{-\!\circ}u' \mid u'(\widetilde{x}).\overline{v}\,\widetilde{x})$. Thanks to Equation 2, $[\![\widehat{C_{L\ell}}]\!]\big[[\![P]\!]\big] \approx_\ell C_{L\ell}\big[[\![P]\!]\big]$. And, with similar reasoning, the same holds for $Q$. These results allow the proof: ¿From the assumption that $P \approx Q$ we get $\widehat{C_{L\ell}}[P] \mathrel{\dot{\approx}} \widehat{C_{L\ell}}[Q]$. By Proposition 8, $[\![\widehat{C_{L\ell}}[P]]\!] \mathrel{\dot{\approx}} [\![\widehat{C_{L\ell}}[Q]]\!]$. Now we focus on $P$: $[\![\widehat{C_{L\ell}}[P]]\!] = [\![\widehat{C_{L\ell}}]\!]\big[[\![P]\!]_{\widetilde{z}}\big]$ (by definition of $[\![\cdot]\!]$); $\approx_\ell [\![\widehat{C_{L\ell}}]\!]\big[[\![P]\!]\big]$ (by Lemma 7.1); $\approx_\ell C_{L\ell}\big[[\![P]\!]\big]$ (by Equation 2). Doing the same to $Q$ we obtain $C_{L\ell}\big[[\![P]\!]\big] \mathrel{\dot{\approx}} C_{L\ell}\big[[\![Q]\!]\big]$, contradicting $[\![P]\!] \not\approx [\![Q]\!]$ and so proving the result. □

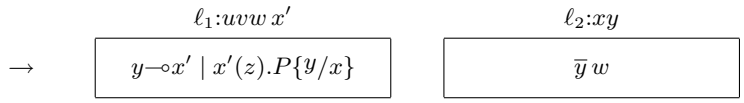
## 5 A linear forwarder machine

In this section we develop a distributed machine for the $L\ell$ calculus, suitable for a point-to-point network such as the Internet. This machine is actually very similar to the fusion machine [**?**], but with linear forwarders instead of fusions (trees of persistent forwarders). We first give a diagrammatic overview of the machine. Then we provide a formal syntax, and prove full abstraction with respect to barbed congruence.

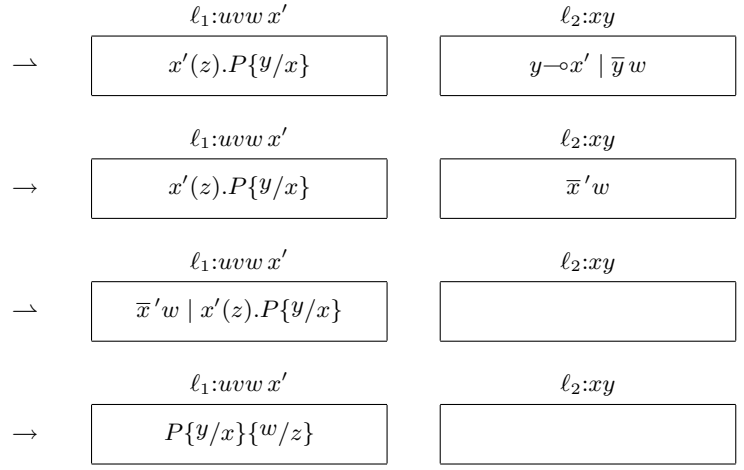
We assume a set of locations. Each channel belongs to a particular location. For instance, channels $u, v, w$ might belong to $\ell_1$ and $x, y$ to $\ell_2$. The structure of a channel-name $u$ might actually be the pair (IP:TCP), giving the IP number and port number of a channel-manager service on the Internet. Every input process is at the location of its subject channel. Output processes may be anywhere. For example,




In a point-to-point network such as the Internet, the output message $\overline{u}\,y$ would be sent to $u$ to react; in a broadcast network such as wireless or ethernet, the offer of output would be broadcast and then $\ell_1$ would accept the offer. In both cases, the result is a reaction and a substitution $\{y/x\}$ as follows:




The overall effect of the linear forwarder $y{-\!\circ}x'$ will be to turn the $\overline{y}\,w$ into $\overline{x}\,'w$. In a point-to-point network this can be implemented by migrating the $y{-\!\circ}x'$ to $\ell_2$, there to *push* the $\overline{y}\,w$ to $x'$, as shown below. (In the following diagrams, some steps are shown as heating transitions $\rightharpoonup$; these steps were abstracted away in the $L\ell$ calculus).



In a broadcast network, the linear forwarder $y{-\!\circ}x'$ would instead stay at $\ell_1$; later, when the offer of $\overline{y}\,a$ is broadcast, the linear forwarder can *grab* the offer.

We remark that the above machine avoids code migration: after an input, the continuation remains in the same place. (With the minor exception that forwarders $x{-\!\circ}x'$ and outputs $\overline{x}\,y$ may migrate; but this is easy to implement). Because code does not migrate, there is no need for a run-anywhere infrastructure such as Java, and it is possible to compile into CPU-specific machine code.

*Distributed choice.* A well-known operator in process calculi is the input-guarded choice. We remark that the distributed version, $x(u).P + y(v).Q$ where $x$ and $y$ are at separate locations, is not even expressible in our machine. However, it can be compiled into a localised input choice as follows:

$$\begin{array}{rcl}[\![x(u).P + y(v).Q]\!] & = & (x'y')\big(\; x{-\!\circ}x' \mid y{-\!\circ}y' \\ & & \qquad \mid\; x'(u).(P \mid y'{-\!\circ}y) + y'(v).(Q \mid x'{-\!\circ}x) \;\big)\end{array}$$

To understand this encoding, note that the new names $x'$ and $y'$ will be created at the same location, and so the choice between $x'$ and $y'$ will a local one. Next, $\overline{x}\,v$ may be forwarded to $x'$, or $\overline{y}\,v$ to $y'$, or both. If the reaction with $x'$ is taken, this yields $y'{-\!\circ}y$ to "undo" the effect of the forwarder that was not taken. (It undoes it up to weak barbed congruence, in the sense that $(y')(y{-\!\circ}y' \mid y'{-\!\circ}y) \approx \mathbf{0}$.) Note that even if the location of $x$ should fail, then the $y$ option remains open; and vice versa.

*Failure.* We briefly comment on the failure model for the linear forwarder machine. It is basically the same as failure in the join calculus: either a message can be lost, or an entire location can fail. If a linear forwarder $x{-\!\circ}y$ should fail, the effect is the same as if a single future message $\overline{x}\,\widetilde{u}$ should fail. A command *'iffail(u) then P'* might be added to determine if $u$'s location is unresponsive. The current theory was largely prompted by criticisms of the fragility of our earlier fusion machine.

### 5.1 The machine calculus

**Definition 11 (Linear forwarder machine)** *Localised linear forwarder machines $M$ are given by the following grammar, where $P$ ranges over terms in the L$\ell$ calculus (Definition 2).*

$$M \quad ::= \quad \mathbf{0} \;\;\big|\;\; x[P] \;\;\big|\;\; (\!|x|\!)[P] \;\;\big|\;\; M, M$$

The presentation here is similar to that given for the fusion machine [**?**]. The *basic channel-manager* $x[P]$ denotes a channel-manager at channel $x$ containing a *body* $P$. The *local channel-manager* $(\!|x|\!)[P]$ denotes a channel-manager where the name $x$ is not visible outside the machine. We write $\mathsf{chan}\,M$ to denote the set of names of all channel-managers in the machine, and $\mathsf{lchan}\,M$ for the names of only the local channel-managers.

We assume a *co-location* equivalence relation $L$ on channels. We write $x@y$ to mean that $(x, y) \in L$, with the intended meaning that the two channels are at the same location. It is always possible to create a fresh channel at an existing location: to this end we assume that each equivalence class in $L$ is infinitely large. In the machine calculus, we generally assume $L$ rather than writing it explicitly.

There are a number of well-formedness conditions on machines:

(1) *Localised.* All code is in the right place, and does not need to be moved at runtime. Formally, in every channel $u[P]$, every free input $v(\widetilde{x}).Q$ satisfies $u@v$. Moreover, no received name is used as the subject of an input.

(2) *Singly-defined.* There is exactly one channel-manager per channel. Formally, a machine $x_1[B_1], \cdots, x_n[B_n]$ is singly-defined when $i \neq j$ implies $x_i \neq x_j$ ($x_i$ or $x_j$ may be local).

(3) *Complete.* It does not make sense to write a program that refers to channels which do not exist. We say that a machine is complete when it has no such references. Formally, the free names of a machine must be contained in $\mathsf{chan}\, M$.

A machine is *well-formed* when it is localised, singly-defined and complete. In the following, we consider only well-formed machines.

**Definition 12 (Dynamics)** *The* structural congruence *for well-formed machines $\equiv$ is the smallest equivalence relation satisfying the following laws:*

$$M, \mathbf{0} \equiv M \qquad M_1, M_2 \equiv M_2, M_1 \qquad M_1, (M_2, M_3) \equiv (M_1, M_2), M_3$$
$$P \equiv Q \quad \textit{implies} \quad u[P] \equiv u[Q] \ \textit{and}\ (\!(u)\!)[P] \equiv (\!(u)\!)[Q]$$

*The reduction step $\rightarrow$ and the heating step $\rightharpoonup$ are the smallest relations satisfying the rules below, and closed with respect to structural congruence. Each rule addresses generically both free and local channel-managers.*

$$u[\overline{u}\,\widetilde{y} \mid u(\widetilde{x}).P \mid R] \;\rightarrow\; u[P\{\widetilde{y}/\widetilde{x}\} \mid R] \qquad \text{(react)}$$
$$u[u{\multimap}v \mid \overline{u}\,\widetilde{x} \mid R] \;\rightarrow\; u[\overline{v}\,\widetilde{x} \mid R] \qquad \text{(fwd)}$$

$$u[(x)P \mid R] \;\rightharpoonup\; u[P\{x'/x\} \mid R],\ (\!(x')\!)[\,], \quad x' \textit{ fresh}, x'@u \qquad \text{(dep.new)}$$
$$u[x{\multimap}y \mid R_1],\ x[R_2] \;\rightharpoonup\; u[R_1],\ x[x{\multimap}y \mid R_2], \quad \textit{if } u \neq x \qquad \text{(dep.fwd)}$$
$$u[\overline{x}\,\widetilde{y} \mid R_1],\ x[R_2] \;\rightharpoonup\; u[R_1],\ x[\overline{x}\,\widetilde{y} \mid R_2], \quad \textit{if } u \neq x \qquad \text{(dep.out)}$$
$$u[x(\widetilde{y}).P \mid R_1],\ x[R_2] \;\rightharpoonup\; u[R_1],\ x[x(\widetilde{y}).P \mid R_2], \quad \textit{if } u \neq x,\ u@x \qquad \text{(dep.in)}$$

*For every transition rule above, we close it under contexts:*

$$\frac{M \rightarrow M', \quad \mathsf{chan}\, M' \cap \mathsf{chan}\, N = \emptyset}{M, N \rightarrow M', N} \qquad \frac{M \rightharpoonup M', \quad \mathsf{chan}\, M' \cap \mathsf{chan}\, N = \emptyset}{M, N \rightharpoonup M', N}$$

We draw attention to two of the rules. The rule *(dep.new)* picks a fresh channel-name $x'$ and this channel is deemed to be at the location where the command was executed. The rule *(dep.in)* will only move an input command from one channel $u$ to another channel $x$, if the two channels are co-located; hence, there is no "real" movement. In the current presentation we have used arbitrary replication

$!P$, but in a real machine we would instead use guarded replication [**?**], as is used in the fusion machine. All rules preserve well-formedness.

**Definition 13 (Bisimulation)** *The observation $M \downarrow u$ is the smallest relation satisfying $u[P] \downarrow u$ if $P \downarrow u$, and $M_1, M_2 \downarrow u$ if $M_1 \downarrow u$ or $M_2 \downarrow u$. Write $\Rightarrow$ for $(\rightarrow^* \rightharpoonup^*)^*$, and $M \Downarrow u$ for $M \Rightarrow \downarrow u$.*

*A weak barbed bisimulation $\mathcal{R}$ between machines is a symmetric relation such that if $M \mathcal{S} N$ then*

1. *$M \Downarrow u$ implies $N \Downarrow u$*
2. *$M \Rightarrow M'$ implies $N \Rightarrow N'$ such that $M' \mathcal{R} N'$*

*Let $\dot{\approx}$ be the largest barbed bisimulation.*

*Two machines $M_1$ and $M_2$ are weak barbed equivalent, written $M_1 \simeq M_2$, when for every machine $N$, then $N, M_1 \dot{\approx} N, M_2$. (Note that $N, M_1$ and $N, M_2$ are assumed to be well-formed, so $\mathsf{chan}\, N \cap \mathsf{chan}\, M_1 = \mathsf{chan}\, N \cap \mathsf{chan}\, M_2 = \emptyset$.)*

The choice of sticking to weak barbed *equivalence* $\simeq$ (which closes under parallel contexts) rather than weak barbed *congruence* $\approx$ (which closes under all contexts) is motivated by the fact that congruence is awkward to define at the machine level. In any case, in the weak asynchronous calculus without the match operator, the two coincide.

We will prove correctness using a translation $\mathsf{calc}\, M = (\mathsf{lchan}\, M)\widehat{M}$ from machines to terms in the $L\ell$ calculus, where

$$\widehat{\mathbf{0}} = \mathbf{0} \qquad \widehat{u[P]} = P \qquad \widehat{(\!|u|\!)[P]} = P \qquad \widehat{M_1, M_2} = \widehat{M_1} \mid \widehat{M_2}$$

One might prefer to prove correctness of a "compiling" translation, which takes a term and compiles it into a machine — rather than the reverse translation $\mathsf{calc}$. However, different compilers are possible, differing in their policy for which location to upload code into. We note that all correct compilers are contained in the inverse of $\mathsf{calc}$, so our results are more general.

The correctness of the forwarder machine relies on the following lemma

**Lemma 14 (Correctness)** *$M_1 \dot{\approx} M_2$ if and only if $\mathsf{calc}\, M_1 \dot{\approx} \mathsf{calc}\, M_2$*

*Proof.* It is clear that machine operations are reflected in the calculus: $M \equiv M'$ implies $\mathsf{calc}\, M \equiv \mathsf{calc}\, M'$, and $M \rightharpoonup M'$ implies $\mathsf{calc}\, M \equiv \mathsf{calc}\, M'$, and $M \rightarrow M'$ implies $\mathsf{calc}\, M \rightarrow \mathsf{calc}\, M'$, and $M \downarrow u$ implies $\mathsf{calc}\, M \downarrow u$.

The reverse direction is more difficult. We wish to establish that

1. $\mathsf{calc}\, M \downarrow u$ implies $M \rightharpoonup^* \downarrow u$, and
2. $\mathsf{calc}\, M \rightarrow P'$ implies $\exists M' : M \rightharpoonup^* \rightarrow M'$ and $P' \equiv \mathsf{calc}\, M'$.

Both parts share a similar proof; we focus on part 2. Given the machine $M$, there is also a *fully-deployed* machine $M'$ such that $M \rightharpoonup^* M'$ and $M'$ has no heating transitions: that is, all unguarded restrictions have been used to create

fresh channels, and all outputs and forwarders are at the correct location. Therefore $\mathsf{calc}\, M \equiv \mathsf{calc}\, M' \to P'$. The structure of $\mathsf{calc}\, M'$ has the form $(\mathsf{lchan}\, M')\widehat{M'}$. The reaction must have come from an output $\overline{u}\,\widetilde{y}$ and an input $u(\widetilde{x}).P$ in $M'$ (or from an output and a forwarder). Because $M'$ is fully-deployed, it must contain $u[\overline{u}\,\widetilde{y} \mid u(\widetilde{x}).P]$. Therefore it too allows the corresponding reaction. The bisimulation result follows directly from the above. □

We now prove full abstraction: that two machines are equivalent if and only if their corresponding calculus terms are equivalent. The bulk of the work is to relate weak barbed *equivalence* $\simeq$ (closure under parallel composition, used in the machine) to weak barbed *congruence* $\approx$ (closure under all contexts, used in the calculus). The two coincide in the weak setting for an asynchronous calculus without match [**?**]: $P \approx Q$ if and only if, for every $R$, then $R|P \mathrel{\dot{\approx}} R|Q$.

It has been suggested that a weaker simulation result would suffice. But we believe that full abstraction shows our machine to be a natural implementation of the pi calculus, in contrast to Facile and Join. Practically, full abstraction means that a program can be purely debugged at source-level rather than machine-level.

**Theorem 15 (Full abstraction)** *$M_1 \simeq M_2$ if and only if $\mathsf{calc}\, M_1 \approx \mathsf{calc}\, M_2$.*

*Proof.* The reverse direction is straightforward, because machine contexts are essentially parallel compositions. In the forwards direction, it suffices to prove that for every $R$, $R|\,\mathsf{calc}\, M_1 \mathrel{\dot{\approx}} R|\,\mathsf{calc}\, M_2$. By contradiction suppose the contrary: namely, there exists an $R$ such that the two are not barbed bisimilar. Expanding the definition of $\mathsf{calc}$ we obtain that $R|(\mathrm{lchan}\, M_1)\widehat{M_1} \mathrel{\dot{\not\approx}} R|(\mathrm{lchan}\, M_2)\widehat{M_2}$.

We now show how to construct a machine context $M_R$ such that $M_R, M_1 \mathrel{\dot{\not\approx}} M_R, M_2$, thus demonstrating a contradiction. Without loss of generality, suppose that $R$ does not clash with the local names $\mathrm{lchan}\, M_1$ or $\mathrm{lchan}\, M_2$. This gives $(\mathrm{lchan}\, M_1)(R|\widehat{M_1}) \mathrel{\dot{\not\approx}} (\mathrm{lchan}\, M_2)(R|\widehat{M_2})$. In order to ensure well-formedness of $M_R, M_1$ and $M_R, M_2$, let $\widetilde{z} = \mathrm{chan}\, M_1 \cup \mathrm{chan}\, M_2$. By Lemma 7 we get $[\![R]\!]_{\widetilde{z}} \approx_\ell R$, and by definition $[\![R]\!]_{\widetilde{z}}$ contains no inputs on $\widetilde{z}$, so satisfying the localised property. Now assume without loss of generality that $R$ contains no top-level restrictions. Let $M_R = u[R]$ for a fresh name $u$ such that, for every free input $u_i(\widetilde{x}).R'$ in $R$, then $u_i@u$. Hence $\mathrm{lchan}\, M_R = \emptyset$ and $\widehat{M_R} = [\![R]\!]_{\widetilde{z}}$. This yields $\mathsf{calc}\, M_R, M_1 = (\mathrm{lchan}\, M_1)([\![R]\!]_{\widetilde{z}}|\widehat{M_1}) \approx_\ell (\mathrm{lchan}\, M_1)(R|\widehat{M_1})$, and similarly for $M_2$. And finally, by construction, both $M_R, M_1$ and $M_R, M_2$ are singly-defined and complete. □

## 6 Further issues

The point of this paper is to implement the pi calculus input capability. We have shown that just a limited form of input capability (where linear forwarders are allowed) is enough to easily express the full pi calculus input capability. We have expressed this formally through a calculus with linear forwarders, and a proof of its full abstraction with respect to the pi calculus encoding.

The calculus in this paper abstracts away from certain details of implementation (such as the choice between a point-to-point or broadcast network). Nevertheless, thanks to its *localisation* property, it remains easy to implement.

*Coupled bisimulation.* There is an interesting subtlety in the encoding of input capability. Our first attempt at an encoding gave, for example,

$$[\![u(x).(x().P \mid x().Q)]\!] \quad = \quad u(x).(x')(x{\multimap}x' \mid x'().P \mid x{\multimap}x' \mid x'().Q)$$

That is, we tried to reuse the same local name $x'$ for all bound inputs. But then, a subsequent reaction $x{\multimap}x' \mid \overline{x}\, z \to \overline{x}\,' z$ would be a commitment to react with one of $x'().P$ or $x'().Q$, while ruling out any other possibilities. This *partial* commitment does not occur in the original pi calculus expression, and so the encoding did not even preserve behaviour. An equivalent counterexample in the pi calculus is that $\tau.P|\tau.Q|\tau.R$ and $\tau.P|\tau.(\tau.Q|\tau.R)$ are not weak bisimilar. We instead used an encoding which has a fresh channel for each bound input:

$$[\![u(x).(x().P \mid x().Q)]\!] \quad = \quad u(x).\big( (x')(x{\multimap}x' \mid x'().P) \mid (x'')(x{\multimap}x'' \mid x''().Q) \big)$$

Now, any reaction with a forwarder is a *complete* rather than a partial commitment. In fact, both encodings are valid. The original encoding, although not a bisimulation, is still a *coupled bisimulation* [**?**]. (Coupled bisimulation is a less-strict form of bisimulation that is more appropriate for an implementation, introduced for the same reasons as here.) In this paper we stuck to the normal bisimulation and the repaired encoding, because they are simpler.

*The join calculus and forwarders.* We end with some notes on the difference between the join calculus [**?**] and the $L\ell$ calculus presented here. The core join calculus is

$$P \;::=\; \mathbf{0} \quad \big| \quad \overline{x}\,\widetilde{u} \quad \big| \quad P|P \quad \big| \quad \text{def } x(\widetilde{u})|y(\widetilde{v}) \rhd P \text{ in } Q$$

The behaviour of the *def* resource is, when two outputs $\overline{x}\,\widetilde{u}'$ and $\overline{y}\,\widetilde{v}'$ are available, then it consumes them to yield a copy $P\{\widetilde{u}'\widetilde{v}'/\widetilde{u}\widetilde{v}\}$ of $P$. Note that $x$ and $y$ are bound by *def*, and so input capability is disallowed by syntax. The core join calculus can be translated into the pi calculus (and hence $L\ell$) as follows [**?**]:

$$[\![\mathbf{0}]\!] \;=\; \mathbf{0} \qquad [\![\overline{x}\,\widetilde{u}]\!] \;=\; \overline{x}\,\widetilde{u} \qquad [\![P|Q]\!] \;=\; [\![P]\!] \mid [\![Q]\!]$$
$$[\![\text{def } x(\widetilde{u})|y(\widetilde{v}) \rhd P \text{ in } Q]\!] \;=\; (xy)(\,[\![Q]\!] \mid !x(\widetilde{u}).y(\widetilde{v}).[\![P]\!]\,)$$

If a join program is translated into the linear forwarder machine and then executed, then the result has exactly the same runtime behaviour (i.e. same number of messages) as the original join program. Additionally, we can provide the same distribution of channels through the co-location operator discussed above.

A reverse translation is more difficult, because of linear forwarders. One might try to translate $x{\multimap}y \mid R$ into def $x(\widetilde{u}) \rhd \overline{y}\,\widetilde{u}$ in $[\![R]\!]$, analogous with the translation of a forwarder into the pi calculus that was used in Proposition 10. But the $L\ell$

calculus allows a received name to be used as the source of a forwarder, as in $u(x).(x{-\!\circ}y \mid P)$, and the same is not possible in the join calculus. Therefore contexts in the $L\ell$ calculus are strictly more discriminating than contexts in the join calculus. (As an example, def $x(u) \rhd \overline{y}\,u$ in $\overline{z}\,x$ is equivalent to $\overline{z}\,y$ in the join calculus, but the context $z(a).(a{-\!\circ}b \mid \overline{a} \mid \_)$ can distinguish them in the $L\ell$ calculus.)

# A Proofs in Section 4

The proofs in Section 4 admit much simpler alternatives — via the property $P \approx_\ell [\![P]\!]$, which is deduced directly from Lemma 7 and the definition of $[\![\cdot]\!]$. However, this property relates terms from two different sub-calculi of the linear forward calculus. The proofs in Section 4 were chosen to avoid all such mixed relations, for sake of familiarity. Similarly, given a localised term $C\big[[\![P]\!]_x\big]$ one cannot deduce that $C\big[[\![P]\!]\big]$ is localised, and so the two cannot be related by $\approx$ (which was defined only on localised terms). However, they are related by $\approx_\ell$.

Boreale [**?**], on the other hand, uses more complicated proofs. The difference is that our encoding $[\![P]\!]_x$ merely indicates that each input use of $x$ must be encoded via a local channel $(x')(x{-\!\circ}x' \mid \ldots)$; Boreale uses a protocol instead of linear forwarders, and his encoding $\{|P|\}_{x'/x}$ indicates that inputs on $x$ will use the protocol on the free channel $x'$, and so the protocol is observable in $\{|P|\}_{x'/x}$ but not in $\{|P|\}_\emptyset$. The effect is that Lemma 7.1 does not hold for him, and so longer proofs of Lemma 7.3 are needed. Additionally, our encoding causes forwarders to be consumed after reaction; Boreale's causes them to be produced by reaction, so requiring the use of an expansion preorder rather than our simpler labelled bisimulation.

**Proposition 8.** For $P, Q$ in the pi calculus, $P \mathrel{\dot\approx} Q$ if and only if $[\![P]\!] \mathrel{\dot\approx} [\![Q]\!]$.

*Proof.* This is similar to Boreale's Proposition 3.5 and 3.6. We need two lemmas:

1. $P \xrightarrow{\mu} P'$ implies $[\![P]\!]_{\widetilde{z}} \stackrel{\mu}{\Longrightarrow} \approx_\ell [\![P']\!]_{\widetilde{z}}$
2. $[\![P]\!]_{\widetilde{z}} \xrightarrow{\mu} P_1'$ implies $P \stackrel{\mu}{\Longrightarrow} P'$ and $P_1' \approx_\ell [\![P']\!]_{\widetilde{z}}$

For Part 1 the cases of $\mu = (\widetilde{v})\overline{u}\,\widetilde{y}$ and $\mu = u(\widetilde{x})$ are straightforward inductions on the derivation of $\xrightarrow{\mu}$. We draw attention to the case $u(x).P \xrightarrow{u(\widetilde{x})} P$, since this relates specifically to linear forwarders. Suppose $u \in \widetilde{z}$. Then

$$\begin{aligned} [\![u(\widetilde{x}).P]\!]_{\widetilde{z}} &= (u')(u{-\!\circ}u' \mid u'(\widetilde{x}).[\![P]\!]_{\widetilde{z}\widetilde{x}}) \\ &\xrightarrow{u(\widetilde{x})} (u')(\overline{u}\,'\widetilde{x} \mid u'(\widetilde{x}).[\![P]\!]_{\widetilde{z}\widetilde{x}}) \end{aligned} \tag{3}$$

and by Lemma 7, this is $\approx_\ell [\![P]\!]_{\widetilde{z}}$. For $\mu = \tau$ we will write out the interesting case, of communication. Suppose $P|Q \xrightarrow{\tau} (\widetilde{v})(P'\{\widetilde{y}/\widetilde{x}\}|Q'$ with $P \xrightarrow{u(\widetilde{x})} P'$ and $Q \xrightarrow{(\widetilde{z})\overline{u}\,\widetilde{y}}$

$Q'$. By the induction hypothesis and Lemma 7.1, $[\![P]\!]_{\widetilde{z}} \stackrel{u(\widetilde{x})}{\Longrightarrow} P_1' \approx_\ell [\![P']\!] \approx_\ell [\![P']\!]_{\widetilde{z}}$ and $[\![Q]\!]_{\widetilde{z}} \stackrel{(\widetilde{v})\overline{u}\,\widetilde{y}}{\Longrightarrow} Q_1' \approx_\ell [\![Q']\!]_{\widetilde{z}}$. Therefore,

$$\begin{aligned}
[\![P|Q]\!]_{\widetilde{z}} &\stackrel{\tau}{\Longrightarrow} \big((\widetilde{v})(P_1'\{\widetilde{y}/\widetilde{x}\} \mid Q_1'\big) \\
&\approx_\ell \ (\widetilde{v})\big([\![P']\!]_{\widetilde{z}}\{\widetilde{y}/\widetilde{x}\} \mid [\![Q']\!]_{\widetilde{z}}\big) \\
&= \ [\![(\widetilde{v})\big(P'\{\widetilde{y}/\widetilde{x}\}|Q'\big)]\!]_{\widetilde{z}}.
\end{aligned}$$

Part 2 is similar. Again, we draw attention to the input case $[\![u(\widetilde{x}).P]\!]_{\widetilde{z}}$. Suppose $u \in \widetilde{z}$. Then $[\![u(\widetilde{x}).P]\!]_{\widetilde{z}} \stackrel{u(x)}{\longrightarrow} \approx_\ell [\![P]\!]_{\widetilde{z}}$ as in Equation 3. The un-encoded term also makes a matching transition: $u(\widetilde{x}).P \stackrel{u(\widetilde{x})}{\longrightarrow} P$, so completing the proof. □