The Fusion Machine
(extended abstract)

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

[1] Department of Computing, Imperial College, London
[2] Department of Computer Science, University of Bologna

Abstract. We present a new model for the distributed implementation of pi-like calculi, which permits strong correctness results that are simple to prove. We describe the *distributed channel machine* – a distributed version of a machine proposed by Cardelli. The distributed channel machine groups pi processes at their channels (or locations), in contrast with the more common approach of incorporating additional location information within pi processes. We go on to describe the *fusion machine*. It uses a form of concurrent constraints called *fusions* – equations on channel names – to distribute fragments of these processes between remote channels. This fragmentation avoids the movement of large continuations between locations, and leads to a more efficient implementation model.

1 Introduction

The pi calculus and its variants have made a significant impact in research on concurrency and distribution. However, we are aware of only two distributed implementations of the pi calculus: Facile [8] which uses a hand-shake discipline for communication, and an indirect encoding into the join calculus [5] which is then implemented on Jocaml [4]. Other related implementations [20, 6] add explicit location constructs to the pi calculus and use different mechanisms for distributed interaction.

There are two reasons for why pi calculus interaction has not been used for distributed interaction. First, synchronous rendezvous (as found in the pi calculus) seemed awkward to implement. Second, a model of distribution has generally been assumed, in which processes are grouped together at locations and use separate mechanisms for distributed versus local interaction. We propose a different distribution model, the *fusion machine*, which avoids the difficulty in synchronous rendezvous and leads to a simple connection between implementation and pi-like calculi.

In our distribution model, each channel exists at its own location (or is co-located with other channels). Each atomic process waiting to rendezvous over a channel is placed directly at that channel (location); thus, synchronous rendezvous is local. After rendezvous, a process' continuation is broken down into atomic processes. These are sent on to their appropriate channels, and are then ready to perform their own subsequent rendezvous. The task of breaking down and sending is known as *heating* [2], and amounts to a directed implementation of structural congruence. In some ways, our model can be regarded as a distributed version of the single-processor model first described by Cardelli [3] and subsequently used in Pict [14, 17].

The fusion machine is distributed over channels, as outlined above. It also uses explicit fusions [7], a form of concurrent constraints on channel names which it implements with trees of forwarders between channels. These explicit fusions enable atomic processes to be fragmented, so avoiding the movement of large continuations between channels. The issue of fragmentation did not arise in the single-processor channel machine used in Pict. In this machine, after a program is involved in rendezvous, a *pointer* to its continuation is sent on to another channel for subsequent rendezvous. In contrast, the (distributed) fusion machine would require the entire continuation to be sent between channels. This explains why fragmententation becomes relevant for a distributed implementation. We treat fragmentation formally by showing that a calculus with limited continuations – the *explicit solos calculus* – is as expressive as the full calculus with continuations. This builds upon earlier results in [13, 10].

The differences between our model and that of Facile and Jocaml are as follows. Facile uses two classes of distributed entities: (co-)located processes which execute, and channel-managers which mediate interaction. This forces it to use a hand-shake discipline for rendezvous. Jocaml simplifies the model by combining input processes with channel-managers. However, it uses a quite different form of interaction, which does not relate that closely to pi calculus rendezvous. It also forces a coarser granularity, in which every channel must be co-located with at least one other. Like Jocaml, the fusion machine combines processes with channel-managers. Unlike Jocaml, our machine has finer granularity and uses the same form of interaction as the pi calculus.

To conclude the paper, we introduce a formal technique to argue about the efficiency of our fusion machine in terms of the number of network messages required to execute a program. As an example, we quantify the efficiency impact of our encoding of continuations into the explicit solos calculus: it does no worse than doubling the total message count.

The structure of the paper is as follows. Section 2 describes a distributed version of the channel-machine, which is closely connected to the pi calculus. Section 3 presents the fusion machine, which is closer to the explicit fusion calculus and solos calculus. Section 4 gives it a formal theory, and includes full abstraction results. Section 5 adds a model of co-location to the machine, and uses this in a proof of efficiency.

2 The Distributed Channel Machine

Cardelli described an abstract machine for synchronous rendezvous which runs in a single thread of execution, in a shared address space. It contains *channel-managers*, each of which contains pointers to programs; these programs are waiting to rendezvous on the channel. It also contains a *deployment bag* of pointers to programs ready to be executed. The mode of operation of the machine is to move pointers between the channels and the deployment bag.

To make it distributed, we instead assume a collection of located channel-managers which run in parallel and which interact. Each channel-manager has its own thread of execution, its own address space and its own deployment bag. The mode of operation of a channel-manager is either to send some fragments across the network to other channel-managers, or to execute other fragments locally.

Assume a set of channel names ranged over by u, v, w, x, y, z. These might be Internet Protocol numbers and port numbers. At each location there is a channel-manager for exactly one channel name. We therefore identify locations, channels and names. Each channel-manager is made from two parts: atoms (A) which are waiting to rendezvous on the channel, and a deployment bag (D) of terms ready to be executed. We picture it as follows:

u	name of this channel-manager
A	atoms
D	deployment bag

The *atoms* are a collection of output atoms $\mathsf{out}x.P$ and a separate collection of input atoms $\mathsf{in}(x).P$. In general they may be polyadic (communicating several names); but in this section we stick to monadic (single names) atoms for simplicity. The *deployment bag* is a multiset of terms – in this section, terms in the pi calculus.

As an example, the following machine is one representation of the program $\overline{u}\,x.P \mid u(y).Q \mid \overline{v}\,x.R \mid v(y).S$. Observe that each action $\overline{u}\,x.P$ is represented either as an atom at location u, or in any location's deployment bag.

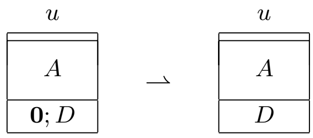
u	v
$\mathsf{out}x.P$ $\mathsf{in}(y).Q$	$\mathsf{in}(y).S$
$\overline{v}\,x.R$	–

There are two kinds of transition for each channel-manager. First, two matching atoms at the same location can *react*:

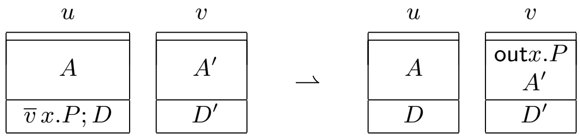
u		u	
$\mathsf{out}x.P$ $\mathsf{in}(y).Q; A$	$\rightarrow$	A	(react)
D		$D; P; Q\{x/y\}$	

(If a replicated input or output atom were involved in the reaction, then a copy of it would be left behind.) Second, program fragments from the deployment bag might be deployed. This is also called *heating* in the process calculus literature.

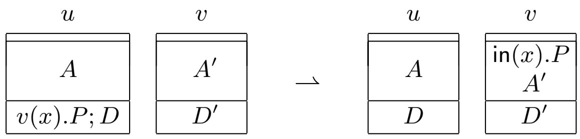
u		u	
A	$\rightharpoonup$	A	(dep.par)
$P \mid Q; D$		$D; P; Q$	



(dep.nil)



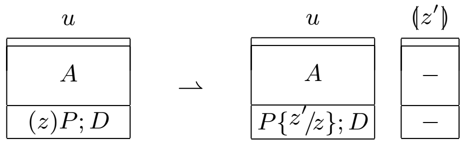
(dep.out)



(dep.in)

These heating transitions are all straightforward. They take a program fragment from the deployment bag, and either break it down further or send it to the correct place on the network. Cardelli's non-distributed machine uses similar rules with minor differences: it uses just a single deployment bag shared by all channel-managers; and because it uses a shared address space, it merely moves pointers rather than entire program fragments.

As for the restriction operator $(z)P$, it has three roles. First, it is a command which creates a new, globally unique channel – a *fresh* name. Second, through rules for alpha-renaming and scope extrusion, it indicates that an existing name should be understood to be globally unique, even though it might be syntactically written with a non-unique symbol. This second role is not relevant to an implementation. Third, it indicates that an existing channel is private, so that a separately-compiled program cannot refer to it by name. For example, it might mean that a machine is not listed in the Internet's Domain Name Service. We will write $(\!|z|\!)$ to indicate a channel z that is not listed. The deployment of restrictions is as follows:



z' fresh (dep.new)

Theorem 1 (Full abstraction) *Two programs are (strongly barbed) congruent in the pi calculus if and only if they are (strongly barbed) congruent in the distributed channel machine.*

This straightforward theorem holds for both the single-processor channel machine and the distributed channel-machine, but as far as we know it has not been given before in the literature. (Cardelli's description of the channel machine anticipated the pi calculus by several years.) The proof is omitted, and will be provided in the full paper. Sewell has given a weaker result for the version of the machine used in Pict [15].

We remark that the full abstraction result for the join calculus is weaker than Theorem 1. This is because the join calculus encodes each pi channel with two join calculus channels that obey a particular protocol. Without a firewall, an encoded program would be vulnerable to any context which violates the protocol. Technically, the join calculus encoding is *non-uniform* as defined by Palamidessi [11]. As for the channel machine, we encode a pi calculus term P by deploying it in a dummy machine $x[P]$. Strictly speaking this is also a non-uniform encoding – but we could make it uniform by adding a structural rule $x[P], x[Q] \equiv x[P; Q]$. Such a rule would be usual in a calculus, but is not relevant in an implementation where different machines have different names by construction. Therefore we do not use it.

Efficiency of Continuations

This distributed version of the channel machine suffers from an efficiency problem. Consider for example the program $\overline{u}.\overline{v}.\overline{x}.\overline{y} \mid u.v.x.y.P$. In the machine, the continuation P would be transported first to u, then v, then x, then y. This is undesirable if the continuation P is large.

There have been two encodings of the pi calculus into a limited calculus without nested continuations. These might solve the efficiency problem. The first encoding, by Parrow [13], uses a sub-calculus of the pi calculus consisting of *trios* so that, for instance, $u(y).\overline{v}\,y$ becomes $t_1(\widetilde{x}).u(y).\overline{t}_2\widetilde{x}y \mid t_2(\widetilde{x}y).\overline{v}\,y.\overline{t}_3\widetilde{x}y$. Here, triggers t_1, t_2, t_3 guard each input and output command, and also transport the entire environment to every continuation. An encoded term could then be executed directly on the distributed channel machine.

The second encoding is based upon the *fusion calculus* of Parrow and Victor [12], a calculus in which the input command $u\widetilde{y}.P$ is not binding. The encoding [10] uses the sub-calculus with only *solos* $\overline{u}\,\widetilde{x}$ and $u\,\widetilde{x}$. It uses the reaction relation

$$(\widetilde{z})(\overline{u}\,\widetilde{x} \mid u\widetilde{y} \mid R) \quad\rightarrow\quad R\sigma$$

where every equivalence class generated by $\widetilde{x} = \widetilde{y}$ has exactly one element not in $\widetilde{z}$, and the substitution σ collapses each equivalence class to its one element.

A single-processor implementation of solos has been described [9]. However, it seems difficult to make a distributed implementation. This is because its reaction is not local: the channel-manager at u must look in the global environment to find sufficient names $(\widetilde{z})$ before it can allow reaction. Instead, we implement the solos calculus with the *explicit fusions* [7]. This allows local reaction as follows:

$$\overline{u}\,\widetilde{x} \mid u\,\widetilde{y} \mid R \quad\rightarrow\quad \widetilde{x}{=}\widetilde{y} \mid R.$$

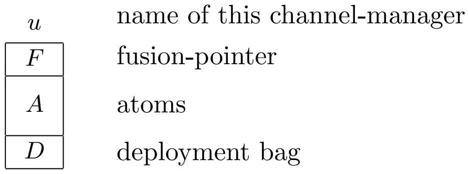
The term $\widetilde{x}{=}\widetilde{y}$ is called an explicit fusion. It has delayed substitutive effect on the rest of the term R. In this respect it is similar to explicit *substitutions* [1]. As an example, in $\overline{u}\,x \mid v\,y \mid u{=}v$, the atom on u may be renamed to v. This yields $\overline{v}\,x \mid v\,y \mid u{=}v$. In contrast to Parrow's trios (which send the entire environment to every continuation), explicit fusions amount to a shared environment.

In fact, we prefer to use terms $\overline{u}\,\widetilde{x}.\phi$ and $u\,\widetilde{x}.\phi$ where ϕ is an explicit fusion continuation – instead of the arbitrary continuations of the channel machine, or the triple continuations of trios, or the empty continuations of the solos calculus. Technically, these fusion continuations allow for an encoding of arbitrary continuations that is uniform and a strong bisimulation congruence (Section 5).

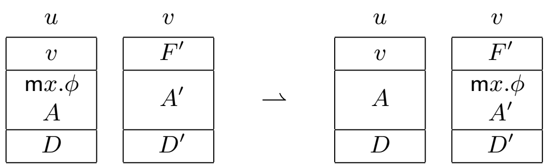
3 Fusion Machine

In general, explicit fusions generate an equivalence relation on names such that any related names may react together. However, in our distributed setting, different names correspond to channel managers at different locations. If two (remote) atoms are related by the equivalence relation, we must send them to a common location in the network so they can react together. The decision as to where to send them must be taken locally. The problem is to find a data structure and an algorithm that allow such local decisions.

The data structure we use to represent each equivalence class is a directed tree. Then each channel can send its atoms to its parent, and related atoms are guaranteed to arrive eventually at a common ancestor. To store this tree, let each channel-manager contain a *fusion pointer* to its parent:

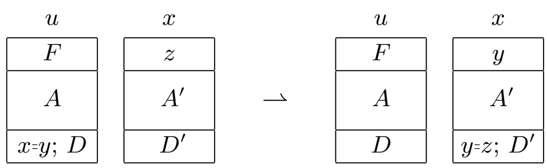


The rule for sending an atom to a parent is called *migration.* (We write m to stand for either in or out).



(migrate)

To update the tree (i.e. to deploy a fusion term), we use a distributed version of Tarjan's *union find* algorithm [16]. This assumes a total order on names, perhaps arising from their Internet Protocol number and port number. The algorithm is implemented with just a single heating rule:

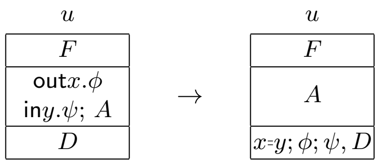


(dep.fu)

where $x < y$ and, if x had no fusion pointer z originally, then we omit $y{=}z$ from the result. This rule amounts to u sending to x the message "fuse yourself to

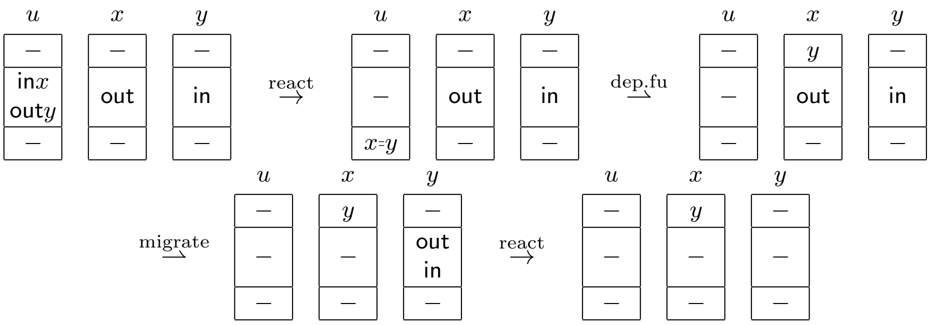
y". To understand this rule, note that it preserves the invariant that the tree of names respects the total order on names, with greater names closer to the root. Therefore, each (dep.fu) transition takes a fusion progressively closer to the root, and the algorithm necessarily terminates. The effect is a distributed, concurrent algorithm for merging two trees.

Finally, we give the modified reaction rule which works with non-binding input and output.



(react)

The following worked example illustrates $u\,x \mid \overline{u}\,y \mid \overline{x} \mid y \rightarrow^* x{=}y$.



We ultimately imagine a hybrid machine which uses both continuations (as in the previous section) and fusions (as in this section). It will use continuations when the continuations are small enough to be efficient; at other times it will use fusions. The two areas are largely unrelated. Therefore, for simplicity, our formal treatment of the machine (next section) omits continuation (and replication). A formal account of the full hybrid machine may be found in [18]. Also, the efficiency results in Section 5 refer to the hybrid machine.

4 Fusion Machine Theory

We now develop a formalism for the fusion machine. We use this to prove that it is a fully abstract implementation of the *explicit solos calculus* (Table 1). For simplicity, we consider the calculus without replication.

We assume a countably infinite set $\mathcal{N}$ of names with a total order, ranged over by u,v,w,x,y,z. Let p range over $\{-\} \cup \mathcal{N}$, denoting pointers including the absent pointer $-$. We use the abbreviation $\widetilde{x}$ for tuples $x_1, \cdots, x_n$, and $\widetilde{x}{=}\widetilde{y}$ for $x_1{=}y_1 | \dots | x_n{=}y_n$. Let ϕ, ψ range over explicit fusions $\widetilde{x}{=}\widetilde{y}$, and m over $\{\mathsf{out}, \mathsf{in}\}$.

Terms P, fusions ϕ, and contexts E in the explicit solos calculus are given by

$$
\begin{array}{lcl}
P & ::= & \mathbf{0} \quad \big| \quad \phi \quad \big| \quad \overline{u}\,\widetilde{x}.\phi \quad \big| \quad u\,\widetilde{x}.\phi \quad \big| \quad (x)P \quad \big| \quad P|P \\
\phi & ::= & \widetilde{x}{=}\widetilde{y} \\
E & ::= & \_ \quad \big| \quad (x)E \quad \big| \quad P|E \quad \big| \quad E|P
\end{array}
$$

Structural congruence on terms $\equiv$ is the smallest congruence satisfying the following:

$$
\begin{array}{l}
P|\mathbf{0} \equiv \mathbf{0} \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)|R \\
x{=}x \equiv \mathbf{0} \quad x{=}y \equiv y{=}x \quad x{=}y \mid y{=}z \equiv x{=}z \mid y{=}z \quad (x)(x{=}y) \equiv \mathbf{0} \\
(x)(y)P \equiv (y)(x)P \quad (x)(P|Q) \equiv (x)P \mid Q \text{ if } x \notin \text{fn}(Q) \\
x{=}y \mid P \equiv x{=}y \mid P\{y/x\}
\end{array}
$$

Reaction relation is the smallest relation $\rightarrow$ satisfying the following, and which is closed with respect to $\equiv$ and contexts: $\overline{u}\,\widetilde{x}.\phi \mid u\,\widetilde{y}.\psi \;\; \rightarrow \;\; \widetilde{x}{=}\widetilde{y} \mid \phi \mid \psi$.

Observation $P \downarrow u$ is the smallest relation satisfying

$$
\begin{array}{lrl}
\overline{u}\,\widetilde{x}.\phi \downarrow u & P \mid Q \downarrow u & \text{if } P \downarrow u \\
u\,\widetilde{x}.\phi \downarrow u & (x)P \downarrow u & \text{if } P \downarrow u \text{ and } u \neq x \\
 & Q \downarrow u & \text{if } Q \equiv P \downarrow u
\end{array}
$$

The *explicit fusion calculus* is obtained by allowing arbitrary continuations and replication: $!P \equiv P|!P$ with $P ::= \ldots \mid \overline{u}\,\widetilde{x}.P \mid u\,\widetilde{x}.P \mid !P$ and $E ::= \ldots \mid \overline{u}\,\widetilde{x}.E \mid u\,\widetilde{x}.E \mid !E$.

Bisimulation is as usual. A relation $\mathcal{S}$ is a strong barbed bisimulation if whenever $P \;\mathcal{S}\; Q$ then

- $P \downarrow u$ if and only if $Q \downarrow u$
- if $P \rightarrow P'$ then $Q \rightarrow Q'$ such that $P' \;\mathcal{S}\; Q'$
- if $Q \rightarrow Q'$ then $P \rightarrow P'$ such that $P' \;\mathcal{S}\; Q'$

Barbed congruence $P \sim Q$ holds whenever, for all contexts E, $E[P] \stackrel{\cdot}{\sim} E[Q]$, where $\stackrel{\cdot}{\sim}$ is the largest bisimulation.

Table 1: The explicit solos calculus

Definition 2 (Fusion machine) *Fusion machines M, bodies B, and terms P are defined by the following grammar:*

$$
\begin{array}{llll}
M & ::= & \mathbf{0} \;\big|\; x[p\colon B] \;\big|\; (\!|x|\!)[p\colon B] \;\big|\; M, M & \text{(machines)}\\
B & ::= & \mathsf{out}\widetilde{x}.\phi \;\big|\; \mathsf{in}\widetilde{x}.\phi \;\big|\; P \;\big|\; B; B & \text{(bodies)}\\
P & ::= & \mathbf{0} \;\big|\; x{=}y \;\big|\; \overline{u}\,\widetilde{x}.\phi \;\big|\; u\,\widetilde{x}.\phi \;\big|\; (x)P \;\big|\; P|P & \text{(terms)}
\end{array}
$$

The *basic channel-manager* $x[p\colon B]$ denotes a channel-manager at channel x containing a fusion pointer to p and a *body* B. This body is an unordered collection of *atoms* $\mathsf{m}\widetilde{x}.\phi$ and *terms* P; it combines the atoms and deployment bag of the previous section. The *local channel-manager* $(\!|x|\!)[p\colon B]$ denotes a channel-manager where the name x is not visible outside the machine. When not relevant, we omit parentheses $(\!|\cdot|\!)$ to address generically channel managers which may be local or global. We also omit the fusion-pointer $x[B]$ to stand for a machine with some unspecified fusion pointer. 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 write $x[\,]$ for $x[\mathbf{0}]$. In terms, the restriction operator $(x)P$ binds x in P; x is free in a term if it occurs unbound. We write $\mathrm{fn}\,P$ to denote the set of free names in P.

There are two well-formedness conditions on machines. First, recall from the previous section that there is exactly one channel-manager per channel. In the calculus, we say that 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). Second, it does not make sense to write a program that refers to a machine which does not exist. We say that a machine is *complete* when it has no such 'dangling pointers'. Formally, define $\mathsf{ptr}\,M$ to be the smallest set containing all free names in all terms in the machine, all non-nil fusion pointers, and all names occurring in any atom $\mathsf{m}\widetilde{x}.\phi$. Then a machine M is complete if $\mathsf{ptr}\,M \subseteq \mathsf{chan}\,M$. A machine is *well-formed* when it is both singly-defined and complete. In the following, we consider only well-formed machines. In particular, when we write $x[P]$ it is shorthand for the (well-formed) machine $x[\text{-}\colon P]$, $y_1[\,]$, $\cdots$ $y_n[\,]$ where $\{y_1, \cdots, y_n\} = \mathrm{fn}(P)\backslash x$. Here, x stands for an arbitrary location where the user of the machine first deploys program P.

Definition 3 (Structural congruence) *The* structural congruence *for machines and atoms $\equiv$ is the least congruence and equivalence satisfying*

1. *Machines and bodies are chemical solutions [2] with $\mathbf{0}$ as unit*
   $M, \mathbf{0} \equiv M \qquad M_1, M_2 \equiv M_2, M_1 \qquad M_1, (M_2, M_3) \equiv (M_1, M_2), M_3$
   $B; \mathbf{0} \equiv B \qquad B_1; B_2 \equiv B_2; B_1 \qquad B_1; (B_2; B_3) \equiv (B_1; B_2); B_3$
2. *Fusion laws*
   $x{=}x \equiv \mathbf{0} \qquad x{=}y \equiv y{=}x$

The fusion laws are a syntactic convenience, allowing us to write a fusion $x{=}y$ without explicitly stating that x and y are distinct names in a particular order. To the same end, we also let $x{=}\text{-}$ stand for $\mathbf{0}$. There is no need to incorporate the calculus congruence $P \equiv Q$ into the machine congruence: the congruence is already implemented by the machine heating transitions.

We remark that the machine has a minimal structural congruence, for ease of implementation: the rule $B_1; B_2 \equiv B_2; B_1$ models the fact that the implementation uses a non-ordered data structure, and so the rule takes no extra work to implement; and $M_1, M_2 \equiv M_2, M_1$ models the fact that machines on the Internet are not ordered. The usual rule $x[B_1], x[B_2] \equiv x[B_1; B_2]$, on the other hand, would not motivated by an implementation: therefore we do not use it.

It is easy to show that all rules in the structural congruence preserve well-formedness.

Definition 4 (Transitions) *The reduction transition $\rightarrow$ and the heating transition $\rightharpoonup$ 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.*

$$u[\mathsf{out}\widetilde{x}.\phi; \mathsf{in}\widetilde{y}.\psi; B] \;\rightarrow\; u[\widetilde{x}{=}\widetilde{y}; \phi; \psi; B] \qquad \text{(react)}$$

$$\begin{array}{rcll}
u[v{:}\, \mathsf{m}\widetilde{x}.\phi; B_1],\ v[B_2] & \rightharpoonup & u[v{:}\, B_1],\ v[\mathsf{m}\widetilde{x}.\phi; B_2] & \text{(migrate)}\\
u[x{=}y; B_1],\ x[p{:}\, B_2] & \rightharpoonup & u[B_1],\ x[y{:}\, y{=}p; B_2], \quad \textit{if } x < y & \text{(dep.fu)}\\
u[v\widetilde{x}.\phi; B_1],\ v[B_2] & \rightharpoonup & u[B_1],\ v[\mathsf{in}\widetilde{x}.\phi; B_2] & \text{(dep.in)}\\
u[\overline{v}\,\widetilde{x}.\phi; B_1],\ v[B_2] & \rightharpoonup & u[B_1],\ v[\mathsf{out}\widetilde{x}.\phi; B_2] & \text{(dep.out)}\\
u[(x)P \mid B] & \rightharpoonup & u[P\{x'/x\}; B],\ (\!| x' |\!)[\text{-:}\ ], \quad x' \textit{ fresh} & \text{(dep.new)}\\
u[P|Q; B] & \rightharpoonup & u[P; Q; B] & \text{(dep.par)}\\
u[\mathbf{0}; B] & \rightharpoonup & u[B] & \text{(dep.nil)}
\end{array}$$

$$\begin{array}{rcll}
u[p{:}\, u{=}y; B] & \rightharpoonup & u[y{:}\, y{=}p; B], \quad \textit{if } u < y & \text{(dep.fu}'\text{)}\\
u[u\widetilde{x}.\phi; B] & \rightharpoonup & u[\mathsf{in}\widetilde{x}.\phi; B] & \text{(dep.in}'\text{)}\\
u[\overline{u}\,\widetilde{x}.\phi; B] & \rightharpoonup & u[\mathsf{out}\widetilde{x}.\phi; B] & \text{(dep.out}'\text{)}
\end{array}$$

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

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

It is easy to show that all transition rules preserve well-formedness. In respect of this, note that *(dep.new)* generates a fresh name so as to preserve single-definition, and the context closure rule forces this name to be globally fresh.

Bisimulation

We now define barbed bisimulation on machines.

Definition 5 (Observation) *The* internal *observation $M \downarrow u$ is the smallest relation closed with respect to structural congruence and satisfying*

$$\begin{array}{rl}
u[\mathsf{m}\widetilde{x}.\phi; B] \downarrow u & \\
u[v{:}\, B], M \downarrow u & \textit{if } M \downarrow v\\
M_1, M_2 \downarrow u & \textit{if } M_1 \downarrow u \textit{ or } M_2 \downarrow u
\end{array}$$

The external *observation $M \Downarrow u$ holds if $M \rightharpoonup^* M'$ such that $M' \downarrow u$ and $u \notin \mathsf{lchan}\, M'$.*

This is standard apart from the middle rule. To understand it, consider the example $u[v\colon\,], v[\mathsf{out}x]$. This corresponds to the calculus term $u{=}v \mid \overline{v}\, x$, which has an observation on u because of the explicit fusion. So too we wish the machine to have an observation on u. As for the reverse case, of $u[v\colon \mathsf{out}x], v[\,]$ being observable on v, this is observable after a single heating transition.

The symbol $\Downarrow$ is generally used for *weak* observation, which is blind to internal reactions. Note however that our external observation $\Downarrow$ is *strong* with respect to reactions, but weak with respect to heating. Similarly, we write $\Rightarrow$ for $\rightharpoonup^*\rightarrow\rightharpoonup^*$.

Definition 6 (Bisimulation) *A (strong) barbed bisimulation $\mathcal{S}$ between machines is a relation such that if $M \;\mathcal{S}\; N$ then*

1. *$M \Downarrow u$ if and only if $N \Downarrow u$*
2. *$M \Rightarrow M'$ implies there exists N' such that $N \Rightarrow N'$ and $M' \;\mathcal{S}\; N'$*
3. *$N \Rightarrow N'$ implies there exists M' such that $M \Rightarrow M'$ and $M' \;\mathcal{S}\; N'$*

Let $\dot{\sim}$, called barbed bisimilarity*, be the largest barbed bisimulation.*

Theorem 7 (Correctness)

1. *For programs P and Q in the explicit solos calculus, $P \;\dot{\sim}\; Q$ if and only if $x[P] \;\dot{\sim}\; x[Q]$.*
2. *There is a translation $(\cdot)^*$ from the pi calculus into the explicit solos calculus such that, for programs P and Q in the pi calculus without replication, $P \;\dot{\sim}\; Q$ if and only if $x[P^*] \;\dot{\sim}\; x[Q^*]$.*

Proof sketch. Consider the translation $\mathsf{calc}\, M$ from machines to terms in the explicit solos calculus, defined by $\mathsf{calc}\, M = (\mathsf{lchan}\, M)[\![M]\!]$ where

$$
\begin{aligned}
[\![\mathbf{0}]\!] &= \mathbf{0} & [\![\mathbf{0}]\!]_u &= \mathbf{0}\\
[\![u[\text{-}\colon B]]\!] &= [\![B]\!]_u & [\![\mathsf{out}\widetilde{x}.\phi]\!]_u &= \overline{u}\,\widetilde{x}.\phi\\
[\![u[v\colon B]]\!] &= u{=}v \mid [\![B]\!]_u & [\![\mathsf{in}\widetilde{x}.\phi]\!]_u &= u\,\widetilde{x}.\phi\\
[\![M_1, M_2]\!] &= [\![M_1]\!] \mid [\![M_2]\!] & [\![P]\!]_u &= P\\
& & [\![B_1; B_2]\!]_u &= [\![B_1]\!]_u \mid [\![B_2]\!]_u
\end{aligned}
$$

It is straightforward to show that machine heating transitions imply structural congruence in the calculus, and that machine barbs and reactions imply barbs and reactions in the calculus.

The proof of the reverse direction is more difficult. Given $\mathsf{calc}\, M \downarrow u$, then there is also a machine M' in which all the deployable terms in M have been deployed such that $\mathsf{calc}\, M' \downarrow u$. We now consider the fusion pointers in M'. Let us write $u \rightsquigarrow v$ if there is a sequence of fusion pointers from u to v. Note that all transitions preserve the following properties of this relation: it is anti-reflexive, anti-symmetric and transitive, it respects the order on names ($x \rightsquigarrow y$ implies

$x < y$) and it is confluent ($x \rightsquigarrow y$ and $x \rightsquigarrow z$ implies $y \rightsquigarrow z$ or $z \rightsquigarrow y$ or $y = z$). We are therefore justified in talking about a tree of fusion pointers. With this tree it is easy to prove that if $\mathsf{calc}\, M' \downarrow u$, then also $M' \downarrow u$. It is a similar matter to show that the machine preserves calculus reactions.

Therefore, the translation calc preserves observation and reaction, and so $\mathsf{calc}\, M \dot{\sim} \mathsf{calc}\, N$ if and only if $M \dot{\sim} N$. The first part of the theorem is just a special case of this, since $\mathsf{calc}\, x[P] \equiv P$.

As for the pi calculus result, we refer to Corollary 66 and Proposition 101 of [18]. Together these provide a translation from the pi calculus into the explicit solos calculus which preserves strong barbed bisimulation. □

We now consider behavioural congruence. In this paper, our goal is that the fusion machine should provide an operational semantics for calculus programs: i.e. we wish to study how programs behave when placed in a machine. To this end, we define contexts E_m for the machine where holes are filled with terms.

Definition 8 (Contexts) *Machine contexts E_m are given by*

$$\begin{array}{lcl} E_m & ::= & x[p\colon E_b] \;\mid\; (\!|x|\!)[p\colon E_b] \;\mid\; E_m, M \;\mid\; M, E_m \\ E_b & ::= & \_ \;\mid\; B; E_b \;\mid\; E_b; B \end{array}$$

When we write $E_m[P]$, we implicitly assume it to be well-formed. The machine equivalence is defined as follows:

Definition 9 (Equivalence) *Two terms are judged equivalent by the machine, $P \sim_m Q$, if and only if for every context E_m, $E_m[P] \dot{\sim} E_m[Q]$.*

Theorem 10 (Full abstraction) *For terms P and Q in the explicit solos calculus, $P \sim Q$ if and only if $P \sim_m Q$.*

Proof sketch. In the forward direction, we extend the translation calc to contexts in the obvious way. Then, given a machine context E_m, we can construct a calculus context $E = \mathsf{calc}\, E_m$ such that, for every P, $\mathsf{calc}(E_m[P]) \equiv E[P]$.

The reverse direction is not so straightforward. Consider for example the context $E = \overline{u}\, x \mid (x)\_$. This has no direct equivalent in the machine: it is impossible in the machine for x to be a local channel-manager whose scope includes a hole, and also at the same time a free name. Instead, given a context E which can discriminate between P and Q, we will construct another context $E' = \overline{u}\, x' \mid (x)\_$ which also discriminates them, and which has no clash of names; therefore it can be represented in the machine.

Technically, we will define a translation $[\![\cdot]\!]_{\widetilde{y}}$ from calculus contexts E to triples $(\sigma, \widetilde{z}, R)$. This translation pushes out the bindings that surround the hole in E. In order to accomplish this structurally, we keep all the binders in $\widetilde{z}$ (suitably renamed to avoid clashes), and collect the necessary renamings of free names in σ. The intention is that for any terms P and Q, then $E[P] \dot{\sim} E[Q]$ if and only if $(\widetilde{z})(R|P) \dot{\sim} (\widetilde{z})(R|Q)$, where $[\![E]\!]_{\widetilde{y}} = (\sigma, \widetilde{z}, R)$ and $\widetilde{y}$ contains all the names occurring in E, P, and Q. The translation is defined as follows:

$$\llbracket \_ \rrbracket_{\widetilde{y}} = (\emptyset, \emptyset, \mathbf{0})$$

$$\llbracket E|S \rrbracket_{\widetilde{y}} = (\sigma, \widetilde{z}, S\sigma|R) \qquad \text{where } \llbracket E \rrbracket_{\widetilde{y}} = (\sigma, \widetilde{z}, R)$$

$$\llbracket (x)E \rrbracket_{\widetilde{y}} = \left\{ \begin{array}{ll} (\sigma[x \mapsto x'], \widetilde{z}x, R) & \text{if } x \notin \widetilde{z} \\ (\sigma[x \mapsto x'], (\widetilde{z}, \sigma(x)), R) & \text{if } x \in \widetilde{z} \end{array} \right\} \quad \begin{array}{l} \text{where } \llbracket E \rrbracket_{\widetilde{y}} = (\sigma, \widetilde{z}, R) \\ \text{and } x' \notin \{\widetilde{y}, \widetilde{z}, \mathsf{ran}\,\sigma\} \end{array}$$

We can prove that the contexts E and $(\widetilde{z})(R|\_)$ are equivalent up to renaming by σ. Since σ is by definition injective, the contexts have the same discriminating power. Hence, so does the machine context $E_m = (\!|z_1|\!)[\,], \ldots (\!|z_n|\!)[\,], x[R; \_]$. □

Unsurprisingly, full abstraction does not also hold for the pi calculus: it is known that pi calculus congruence is not closed with respect to substitution, whereas explicit fusion contexts always allow substitution.

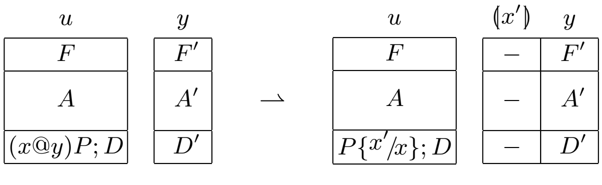
5 Co-location

We now refine the abstract machine with (co-)locations, to allow practical reasoning about efficiency. When two machines are running at the same physical location, and share an address space, we draw their diagrams as physically adjacent:

u	v
F	F'
A	A'
D	D'

Some optimisations are possible in this case. First, it is possible to migrate or deploy an arbitrarily large number of terms to an adjacent machine, in constant time and without requiring any inter-location messages. Second, we can use just a single thread to handle both channels. In the degenerate case, where all machine's channels are at the same location and handled by just a single thread, the machine is essentially the same as Cardelli's single-processor machine.

Co-location might be programmed with a *located restriction* command in the calculus, written $(x@y)P$, to indicate that the new channel x should be created physically adjacent to y. The deployment transition is



(dep.new.at)

(To implement this efficiently, without sending any inter-location messages, we assume that u is able to generate the fresh channel name x' locally – even though that x' will reside outwith u. We could implement this by letting each channel name incorporate a Globally Unique Identifier.)

Note that bound input, as found in the distributed version of the channel machine, allows new names to be created at a location chosen at runtime. For instance, $\mathsf{in}(x).(y@x)P$ will create the name y at the location of whichever name substitutes x. By contrast, a fusion machine without bound input has no way to chose locations at runtime. Therefore, bound input should be retained.

Co-location Used in Encoding Continuations

As discussed in Section 3, we ultimately imagine a machine which uses both continuations and fusions, and which uses them to implement the full pi calculus and explicit fusion calculus with nested continuations. To avoid the cost of repeatedly transporting continuations, an optimising compiler can encode a term with nested continuations into one without. This section describes our encoding and discusses its efficiency.

Two different encodings have been given previously [13, 10]. The distinguishing features of ours are that it is a strong congruence rather than just preserving weak congruence, it is uniform, and it uses co-location for increased efficiency. As an example, we encode the term $\overline{u}.(\overline{v} \mid v)$ as

$$(v'@v, v''@v)(\overline{u}.(v{=}v'{=}v'') \mid \overline{v}' \mid v'').$$

Note that the commands $\overline{v}'$ and v'' will necessarily remain idle until after $\overline{u}$ has reacted. Then, since v' and v'' are co-located with v, it will take no inter-location messages to migrate them to v.

Technically, we will relate terms P to triples of the form $(\widetilde{x}, \phi, P')$. This triple should be understood as the term $(\widetilde{x})(\phi|P')$ in which P' contains no nested actions or unguarded explicit fusions, and the located restrictions $\widetilde{x}$ are alpha-renamable.

Definition 11 *The function* $\mathsf{flat}\,\cdot$ *from terms in the explicit fusion calculus to terms in the explicit solos calculus is as follows. It makes use of an auxiliary translation* $[\![\cdot]\!]$ *from terms in the explicit fusion calculus to triples* $(\widetilde{x}, \phi, P')$*, where* $\widetilde{x}$ *ranges over located restrictions and normal restrictions.*

$$\begin{array}{ll}
[\![\mathbf{0}]\!] = (\emptyset,\ \emptyset,\ \mathbf{0}) & \\
[\![x{=}y]\!] = (\emptyset,\ x{=}y,\ \mathbf{0}) & \\
[\![(z)P]\!] = (z\,\widetilde{x},\ \phi,\ P') & \textit{where } [\![P]\!]{=}(\widetilde{x},\phi,P') \textit{ and } z \notin \widetilde{x} \\
[\![\overline{u}\,\widetilde{z}.P]\!] = \big(u'@u,\ u{=}u',\ (\widetilde{x})(\overline{u}'\widetilde{z}.\phi \mid P')\big) & \textit{where } [\![P]\!]{=}(\widetilde{x},\phi,P'),\ \widetilde{x}\cap\widetilde{z}{=}\emptyset,\ u' \textit{ fresh} \\
[\![u\,\widetilde{z}.P]\!] = \big(u'@u,\ u{=}u',\ (\widetilde{x})(u'\widetilde{z}.\phi \mid P')\big) & \textit{where } [\![P]\!]{=}(\widetilde{x},\phi,P'),\ \widetilde{x}\cap\widetilde{z}{=}\emptyset,\ u' \textit{ fresh} \\
[\![P \mid Q]\!] = (\widetilde{x}\,\widetilde{y},\ \phi|\psi,\ P'|Q') & \textit{where } [\![P]\!]{=}(\widetilde{x},\phi,P'),\ [\![Q]\!]{=}(\widetilde{y},\psi,Q') \\
 & \textit{and } \widetilde{x}\cap(\{\widetilde{y}\}\cup\mathrm{fn}(\psi|Q'))=\emptyset \\
 & \textit{and } \widetilde{y}\cap(\{\widetilde{x}\}\cup\mathrm{fn}(\phi|P'))=\emptyset \\
\mathsf{flat}\,P = (\widetilde{x})(\phi \mid P') & \textit{where } [\![P]\!]{=}(\widetilde{x},\phi,P')
\end{array}$$

Theorem 12 *For any term* P *in the explicit fusion calculus,* $P \sim \mathsf{flat}\,P$*.*

This encoding is only defined on terms without replication. However, the encoding is a congruence even within replicated contexts. For instance, $!\overline{u}\,\widetilde{x}.P \sim !\overline{u}\,\widetilde{x}.(\mathsf{flat}\,P)$. Therefore, an optimising compiler can locally encode any part of a program, without needing to encode it all. The proof is substantial; it may be found in [18]. Other encodings of replication are also possible, in the style of [13] or [9].

Theorem 13 *If $x[P]$ takes n inter-location messages to evolve to M' in the fusion machine with continuations, then $x[\mathsf{flat}\,P]$ needs to take no more than $2n$ inter-location messages in the machine without continuations to evolve to N', such that $M' \mathrel{\dot{\sim}} N'$.*

Proof sketch. First, annotate the machine transitions from Definition 4 with 0 or 1 to indicate their cost. For instance, migration $u[v{:}\,\mathsf{out}x],\, v[\,] \rightharpoonup^0 u[v{:}\,],\, v[\mathsf{out}x]$ takes no messages if u and v are co-located, and one message $\rightharpoonup^1$ otherwise. Then, define a *costed simulation relation* where $P\ \mathcal{S}\ Q$ implies that transitions $P \rightarrow^i P'$ or $P \rightharpoonup^i P'$ can be matched by transitions in Q of cost no greater than $2i$. Construct $\mathcal{S}=\{(M,N)\}$ where for each term P contained in a channel-manager in M, then N contains $\mathsf{flat}\,P$ in any channel-manager. Then $\mathcal{S}$ is a costed simulation. □

6 Conclusions

We have introduced the fusion machine, a distributed channel-based machine for implementing several pi-like calculi such as the synchronous and asynchronous pi calculi, the explicit fusion calculus and the explicit solos calculus. Our objective was to make an easily-implementable machine that corresponds closely to such pi-like calculi. This conjectured ease of implementation appears to be born out in a prototype implementation by Wischik [19] and in projects by students at the University of Bologna. With respect to the close correspondence with pi-like calculi, we have proved abstraction results which are stronger than those obtained for other implementations. On the contrary, the fusion calculus [12] and the solos calculus [10] are awkward to implement in the fusion machine, even though they are closely related to the explicit fusion calculus. This is because they only allow reaction after a global search for restricted names.

We are currently working on a full distributed implementation for the fusion machine, and on a fusion-based language incorporating transactions and failures. We also plan to use some structure richer than just co-location, perhaps to model firewalls or other ambient-like boundaries. We would also like to mention the Xspresso project at Microsoft Redmond – a project to develop a pi-like language with explicit fusions, and a corresponding distributed machine. The machine is similar to the fusion machine presented in this paper, although it implements fusions with an alternative to forwarder-trees – the hope is to make them more scalable, and robust in the presence of failure.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
3. L. Cardelli. An implementation model of rendezvous communication. In *Seminar on Concurrency*, LNCS 197:449–457, 1984.
4. S. Conchon and F. L. Fessant. Jocaml: Mobile agents for objective-caml. In *ASA/MA'99*, pages 22–29. IEEE, Computer Society Press.
5. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL'96*, pages 372–385. ACM Press.
6. C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In *IFIP TCS 2000*, LNCS 1872:348–364.
7. P. Gardner and L. Wischik. Explicit fusions. In *MFCS 2000*, LNCS 1893:373–382.
8. A. Giacalone, P. Mishra, and S. Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.
9. C. Laneve, J. Parrow, and B. Victor. Solo diagrams. In *TACS 2001*, LNCS 2215:127–144.
10. C. Laneve and B. Victor. Solos in concert. In *ICALP'99*, LNCS 1644:513–523.
11. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *POPL'97*, pages 256–265. ACM Press.
12. J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *LICS'98*, pages 176–185. IEEE, Computer Society Press.
13. J. Parrow. Trios in concert. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 621–637. MIT Press, 2000.
14. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
15. P. Sewell. On implementations and semantics of a concurrent programming language. In *CONCUR'97*, LNCS 1243:391–405.
16. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
17. D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.
18. L. Wischik. *Explicit Fusions: Theory and Implementation*. PhD thesis, Computer Laboratory, University of Cambridge, 2001.
19. L. Wischik. Fusion machine prototype. http:// www.wischik.com/ lu/ research/ fusion-machine.
20. P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Computer Laboratory, University of Cambridge, 2000.