Explicit Fusions: Theory and Implementation

Lucian Wischik
http://www.wischik.com/lu
November 2001

Abstract

This work describes a concurrent, distributed abstract machine for the pi calculus. Its primary audience are researchers in the field of concurrency. The secondary audience are working programmers looking for a better way to write interactive programs. This audience need only read Chapters 1 (an overview) and 5 (the implementation). I hope that these chapters might prove appealing to students wishing to implement the pi calculus, perhaps as a course project.

The pi calculus of Milner, Parrow and Walker [47] is a widely studied formalism for describing interactive and concurrent systems. Its basic mechanism is synchronous message-passing over a channel: (1) One program signals its readiness to transmit some data over a channel; (2) Another program signals its readiness to receive over that channel; (3) When it has been established that two programs are ready to communicate, they do.

I introduce a new model for synchronous rendezvous—using *explicit fusions*. 'Fusion' means that, during communication, the data is temporarily shared between the two participating programs. 'Explicit' means that this fusion can persist in the program, allowing us to delay and control the effect of the communication. In this sense, explicit fusions do for the pi calculus what the explicit substitutions of Abadi, Cardelli, Curien and Lévy [1] do for the lambda calculus.

The dissertation has two halves: theory and implementation. The theory introduces the *explicit fusion calculus*—a variant of the pi calculus that includes explicit fusions. I study its bisimulation, and relate it to the fusion calculus of Victor and Parrow [52], the chi calculus of Fu [21] and Sangiorgi's open bisimulation for the pi calculus [59]. Fusions, although not explicit ones, were initially and independently discovered by Victor and Parrow and by Fu. A pi-calculus term called an *equator*, introduced by Honda and Yoshida [31], behaves like an explicit fusion up to weak bisimulation. Merro [40] has studied the connection between equators and the fusion calculus; I study their connection with explicit fusions. Some of the work here has already been published by Gardner and Wischik [25].

The implementation involves a distributed abstract machine for both the pi calculus and the explicit fusion calculus. I introduce a technique called *fragmentation* which leads to more efficient operation of the machine, and show how fragmentation can be encoded in the explicit fusion calculus. This fragmentation is similar to the *solos* calculus of Laneve, Victor and Parrow [34], in that it does not use syntactic guards. Now there have been implementations of the pi calculus before: PICT by Pierce is the best known; Squeak [11] based on a paper by Cardelli [9]; Facile [26]; and the Join Calculus [17] by Fournet, Lévy and others. What distinguishes my work is that it is an distributed virtual machine for the pi calculus which implements synchronous rendezvous without handshaking. The others implementations are either not distributed (PICT, Squeak), or use handshaking (Facile), or do not implement the pi calculus (Join). I discuss optimisations for when various agents share the same address space. In the limiting case when the entire program occupies just a single address space, my implementation becomes essentially the same as PICT and Squeak.

Parts of Chapters 3 and 4 restate existing work [25], done in collaboration with my supervisor Philippa Gardner. She has also contributed thorough proofreading. The machine calculus in Chapter 6 has been improved through the help of her and Cosimo Laneve.

Contents

Chapter 1

Synchronous rendezvous

A *concurrent system* is a collection of programs which run at the same time and interact with each other. Some examples are a computer's operating system, the Internet, and a telephone network. The last two are also *distributed*— the programs run on physically separate machines. In the past few decades of research into concurrent and distributed systems, many theoreticians have focused on one particular form of interaction between programs: *synchronous rendezvous at a channel*. It works as follows.

1. One program signals its intention to transmit some data over a particular channel, and then waits.

2. Another program signals its readiness to receive data over that channel, and also waits.

3. When it has been established that two programs are ready to communicate, they do, and both are then free to continue.

In 1989 Milner, Parrow and Walker proposed a calculus [47] in which programs do nothing but communicate, and the content of their communication is nothing but channel-names. Surprisingly, this calculus is computationally as powerful as a Turing machine. Because of its minimality and completeness, it has attracted much theoretical interest. It is called the *pi calculus*.

Five years ago Victor and Parrow [51, 52] and Fu [21] presented a model of rendezvous where, at the instant of communication, both sender and receiver simultaneously hold the data: it is *fused* between sender and receiver. This means that either program can refer interchangeably to what the sender sent, or to what the receiver received. My thesis is that we can prolong this instant and use it to our advantage. Such a prolonged fusion I call an *explicit fusion*. I develop a theory of explicit fusions, and show how they actually confer two advantages: they allow for a simpler account of fusions, and they make possible a distributed implementation of the pi calculus. They may also have more general application: they partly inspired Milner's current work on minimal reaction contexts [46], for instance.

Synchronous rendezvous is conspicuously different from the basic mechanism of the Internet. In the Internet, the receiver program creates its own local channel and waits on it, the sender program does not wait for successful transmission

but instead continues immediately, and sometimes data gets lost in transit. Synchronous rendezvous is also different from the mechanisms typically provided within a single machine. Here one program places its data in a shared area; another program then looks in the area and finds the data; and the area is protected against concurrent access by a bewildering array of locking commands. In both cases, synchronous rendezvous seems easier to use. The challenge is to implement it on top of the other mechanisms in an efficient and safe way. I show that explicit fusions allow for an implementation of synchronous rendezvous which avoids handshaking and which uses only small messages. And I show that explicit fusions allow for a calculus in which every atomic step in the implementation corresponds to an atomic step in the calculus. This means that, even in the presence of failure, the calculus is still a valid model of the implementation.

In 1985, Cardelli proposed a *uniprocessor* implementation of synchronous rendezvous [9]: that is, an implementation where all the concurrent programs exist on a single machine, and take turns to be executed by a single processor. In 1995, Pierce and Turner described an abstract machine for the pi calculus based on the same technique [55]. This abstract machine was used to build Pict [68, 56, 54], a programming language and compiler for the pi calculus. Turner left open the problem of a distributed abstract machine for the pi calculus. Now one such distributed machine had already been developed in 1989 as part of Facile [26], a language which combines synchronous rendezvous with the lambda calculus. However, Facile's distributed operation involves handshaking and so is not atomic. Other researchers have not further addressed the problem of distributed synchronous rendezvous. Instead, most invent new communication commands for remote interaction, perhaps retaining the pi calculus but only for local interaction on a single processor. Following this approach is the distributed pi calculus of Hennessy and Riely [57], the receptive distributed pi of Amadio and Boudol [4], the ambient calculus of Cardelli and Gordon [10], and Sewell's nomadic pi calculus [64]. The new commands usually involve *asynchronous* rendezvous, where sender programs have no continuation after the rendezvous. All these distributed calculi have the same motivation for inventing their new remote commands: it seemed too hard to make the synchronous rendezvous both atomic and efficient in the presence of message loss. The same motivation led Fournet and Levy to develop the join calculus [18, 17, 16]: this uses a somewhat different communication mechanism, but one that is easier to implement in a distributed system. The join calculus has been implemented by Fournet, Fesson and others [35]. Fournet, Schmitt and Levy have implemented the ambient calculus by translating it into the join calculus [19]. Meanwhile, Sewell and Wojciechowski have implemented the nomadic pi calculus by augmenting Pict with some new distributed features.

In this dissertation I present the first concurrent and distributed abstract machine for the pi calculus itself. This machine fragments programs and distributes them between physically remote computers. And, crucially, it implements synchronous rendezvous in a new way—with explicit fusions. For this reason I call it the *fusion machine*. This chapter surveys the related work in more detail, and indicates how and why the fusion machine differs from existing implementations.

1.1 The pi calculus

We start with a brief tutorial in the pi calculus. This section is presented by example; mathematically inclined readers may instead prefer the formal definition given in Section 4.4 (page 67). Two books provide a more complete account: *Communicating and Mobile Systems* by Milner [45], and *The pi calculus* by Sangiorgi and Walker [60].

The pi calculus is a language for describing interactive, concurrent systems. Programs in it are *dynamic*, able to generate new states and new sub-programs. They are also *reconfigurable*, able to modify the connections between each other. One example application is a cellular phone network: when a phone moves from one cell to another, the connection with the previous base station must be broken, and a new connection made to the new base station. Thus, the system reconfigures itself. Another example is an interactive drawing program: when the user selects an object, selection-handles are created; when the user de-selects that object, the handles are destroyed; and the resizing handles have one set of states and behaviours while the rotation handles have another. Thus, the system dynamically creates new states and behaviours. We illustrate the pi calculus through a collection of less ambitious examples.

The pi calculus describes programs which run in parallel and which interact over channels. Consider the example term

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

It contains one program $\overline{u}x.P$ ready to output the data x over the channel u, and afterwards to continue doing P. The output command is said to *guard* the continuation P. In parallel, a second program $u(y).Q$ is ready to receive the data y over the channel, and afterwards continue doing Q. Here the name y is a formal argument and is local to Q. Their rendezvous is as follows:

$$\overline{u}x.P \mid u(y).Q \longrightarrow P \mid Q\{x/y\}.$$

In the pi calculus, the things that are sent and received in communication are channel names. Mainly in this dissertation I use u and v to name channels when they are used in rendezvous, and x and y to name channels when they are sent as data, but there is no difference between the channels themselves. The notation $\overline{u}.P$ stands for a program which sends an empty message on channel u before continuing, and $\overline{u}\tilde{x}$ on its own stands for a program with no continuation.

When terms have essentially the same physical structure, we say they are *structurally congruent*. For instance, two parallel programs may be written in any order:

$$P \mid Q \equiv Q \mid P.$$

To declare a name x that is local in program P, we use the restriction operator:

$$(x)P.$$

Note that input $u(x).P$ and restriction $(x)P$ are different operators. They both use parentheses to indicate that x is local in P.

Names are more like heap variables than stack variables: they can be used outside the sub-program in which they were first created, and can persist after the sub-program ends; whereas stack variables can do neither. Laws for scope extrusion indicate just how far a name might have escaped. In the following, the rendezvous at u will allow x to be used inside Q (assuming that some other x is not already mentioned in Q). For simplicity, we write scope extrusion as an equivalence that happens before the reaction, rather than a part of reaction:

$$\begin{aligned}(x)\overline{u}x.P \mid u(y).Q &\equiv (x)(\overline{u}x.P \mid u(y).Q) \\ &\longrightarrow (x)(P \mid Q\{x/y\})\end{aligned}$$

Terms may be *replicated*. This allows programs to express infinite behaviours, accomplishing the same end as recursion and while loops. Replication is accounted for by the structural congruence: a replicated term is 'equivalent' to many copies of itself.

$$\overline{u}x \mid !u(y).\overline{y} \equiv \overline{u}x \mid u(y).\overline{y} \mid !u(y).\overline{y} \longrightarrow \overline{x} \mid !u(y).\overline{y}$$

Defined in this way, replication makes for a convenient algebra. However it is not satisfactory for an implementation, since it allows an unlimited number of copies to be created. Instead, both Pict and the fusion machine implement replication by creating copies on demand (Section 5.4, page 85).

Three further operators are sometimes used in the literature. In fact, even without them, the calculus is already Turing powerful. I recall them here for completeness, but will not use them in the rest of the dissertation. The first operation is *summation*. In the example

$$\overline{u}z \mid \big(u(x).P + u(y).Q\big) \longrightarrow P\{z/x\},$$

the summed program can choose which of its two possibilities to pursue: in this case it chose the first. The other two operators are *match* $[x{=}y]P$ and *mismatch* $[x{\neq}y]P$. The first blocks until x and y become equal; the second starts unblocked, but becomes blocked when x and y become equal. Consider the following example:

$$\overline{u}x \mid \overline{u}y \mid \overline{u}z \mid u(w).\big([w{=}x]P \mid [w{=}y]Q \mid [w{\neq}x]R\big).$$

Here, depending on which of the three outputs managed to interact, three results are possible: either a substitution $\{x/w\}$ allowing just P to continue, or a substitution $\{y/w\}$ allowing Q and R to continue, or a substitution $\{z/w\}$ allowing just R to continue. Mismatch is a puzzling operator. It sometimes creates a *now or never* situation, since a mismatch $[x{\neq}y]P$ might be satisfied at one instant, but x might become permanently fused to y at the next.

1.2 Pi calculus variations

There are almost as many variations of the pi calculus as there are researchers working on it. Some variations add new constructs. For instance, the spi calculus of Abadi and Gordon [3] adds cryptography so as to support reasoning about concurrent security protocols; and Sewell's already-mentioned nomadic pi calculus adds distribution to support reasoning about concurrent distributed

protocols. Other variations modify the basic mechanism of interaction. One particular variation is relevant to my work: symmetric interaction. In a discipline as fluid as concurrent systems, symmetry at least provides a fixed point. In this section I describe the symmetric versions of the pi calculus that have so far been proposed, and explain how my approach differs.

The pi calculus is asymmetric in the sense that it has an input prefix $u(x)$ that binds, and an output prefix $\overline{u}x$ that does not. Sangiorgi invented a variant called the *private pi calculus* [58] (formally known as pi-I) in which both input and output are binding. An example reaction is

$$\overline{u}(x).P \mid u(y).Q \longrightarrow_{\text{pp}} (z)\big(P\{z/x\} \mid Q\{z/y\}\big). \qquad \textit{Private pi calculus}$$

With the private pi calculus, Sangiorgi addressed the question of whether non-binding output is essential to the expressiveness of the pi calculus. He established that no, it is not.

The pi calculus has two operators that bind—input and restriction—when a more parsimonious calculus is possible in which only one of them binds. The instinct for symmetry with parsimony has led Victor and Parrow to the *fusion calculus* [52] and, independently, Fu to the *chi calculus* [21]. Both use non-binding input. An example reaction is

$$(x)\big(\overline{u}x.P \mid uy.Q \mid R\big) \longrightarrow_{\text{fu}} P\{y/x\} \mid Q\{y/x\} \mid R\{y/x\}. \qquad \textit{Fusion calculus}$$

In this reaction the names x and y have been fused, in the sense that the programs P, Q and R might have referred to x or to y interchangeably. Because it happened to be x that was restricted, we substituted y for x. If instead y had been restricted, then we would have made the reverse substitution x for y. Both the fusion calculus and the chi calculus require that at least one of those names be restricted. More generally, if multiple names are involved in communication, then for each equivalence-class of fused names only a single witness may be unrestricted. For instance,

$$\begin{array}{ll} (x)\big(\overline{u}xy \mid uzz \mid R\big) & \not\longrightarrow_{\text{fu}} \quad \text{Reaction impossible, since } y \text{ and } z \text{ unrestricted.} \\ (xy)\big(\overline{u}xy \mid uzz \mid R\big) & \longrightarrow_{\text{fu}} \quad R\{z/x\}\{z/y\}. \end{array}$$

The fusion calculus was also motivated by a connection with concurrent constraint programming [70]; and the chi calculus by the connection between synchronous rendezvous and cut elimination in proof nets [22]. Fu has studied at length the axiomatisation of the chi calculus without replication, but with match and mismatch [24].

Note that reaction in the fusion and chi calculi is a non-local operation: the channel u cannot allow reaction unless it can find an enclosing restriction operator; and the scope of the fusion affects R as well as P and Q. Because reaction in the fusion calculus is non-local, it is awkward to implement it in a distributed system.

I propose a different treatment of fusion reaction: rather than requiring that all fusions be restricted, we *allow unrestricted fusions to persist as explicit fusions.* For instance,

$$\overline{u}x.P \mid uy.Q \mid R \longrightarrow x{=}y \mid P \mid Q \mid R. \qquad \textit{Explicit fusion calculus}$$

Although this example happens to show an extra program R, the reaction at channel u is local and requires neither R nor an enclosing restriction. This makes it easy to implement. After the reaction, and since a fusion of two names enables them to be used interchangeably, we get the desired substitution:

$$x{=}y \mid P \mid Q \mid R \;\equiv\; x{=}y \mid P\{y/x\} \mid Q\{y/x\} \mid R\{y/x\}.$$

We have seen two calculi with non-binding communication. The first, the fusion calculus, has fusions 'implicit' in its reaction relation. The second, the explicit fusion calculus, has fusions as explicit terms in the calculus. It seems that fusions are a natural consequence of non-binding communication.

On a linguistic note, the phrase *bound name* is perhaps unfortunate for a calculus with non-binding input. *Bound* is a verb meaning 'to encircle or make local'. It is also the past participle of the verb bind, meaning 'to attach'. In the pi calculus, so-called bound input $u(x).P$ both binds and bounds. But in the explicit fusion calculus, restriction $(x)P$ bounds, while input $ux.P$ binds. For this reason, Fu and others prefer to use the phrases *local* and *global* names. In this dissertation I call them *bound* and *free*, for familiarity.

The explicit fusion calculus constitutes one of the two main contributions of this dissertation. I develop it at length in Chapters 2 to 4. The key technical results are as follows.

- (Section 3.8) Strong barbed congruence and strong ground congruence coincide for the explicit fusion calculus.
- (Section 3.5) I provide an efficient characterisation of congruence—that is, a definition which does not involve an infinite quantification over contexts.
- (Section 3.4) I use a new labelled transition, the 'ask' fusion transition, to define the efficient characterisation.
- (Section 3.9) Up to weak barbed bisimulation, equators are like explicit fusions in the sense that they allow names to be interchanged.
- (Section 4.3) Strong congruence in the explicit fusion calculus provides a sound and complete ('fully abstract') model for hyper-equivalence in the fusion calculus.
- (Section 4.5) Strong congruence in the explicit fusion calculus provides a sound model for barbed congruence in the pi calculus.

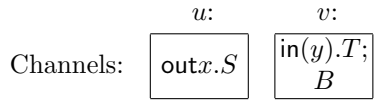
1.3 Uniprocessor pi implementation

In 1985, Cardelli proposed a uniprocessor technique to implement synchronous rendezvous [9]. He and Pike used this technique to implement Squeak [11], a concurrent language for controlling a graphical user interface. In 1994, Pierce and Turner used the same technique to create a uniprocessor abstract machine for the pi calculus [55, 68]. This section briefly describes the technique. However, it is difficult to extend the technique to a distributed abstract machine. The following sections point out the difficulties, and outline my solutions.

One characteristic of a uniprocessor system is its *shared address space*. This means that all names have a constant and small lookup cost. It is therefore is

cheap to exchange data—even complex data—simply by exchanging names or pointers. By contrast, in a distributed system it takes significantly more time to exchange data between, say, www.wischik.com and one's personal computer.

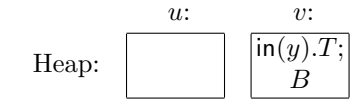
I now describe the uniprocessor abstract machine. A state of the machine consists of a set of program fragments, a set of *channels*, and a queue. Each channel is a set of pointers to program fragments: this indicates that these fragments are currently waiting to send or receive on that channel. Each channel is named, and can be referenced by that name. The queue contains pointers to program fragments: this indicates that these fragments are ready to run, not waiting to send or receive. In general the program fragments may be in a functional language, or an imperative language, or in the pi calculus. An example state is shown below. For clarity we draw the channels and queue as containing actual program fragments, rather than just pointers to them.



Queue: $u(y).P;\ \overline{u}z.Q;\ R$

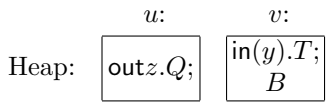
In this example there are three programs $u(y).P$, $\overline{u}z.Q$ and R in the queue, with $u(y).P$ at the head of the queue. There are two channels: channel u contains a program $\mathsf{out}x.S$ waiting to perform output; and channel v contains a program waiting for input, as well as some other program B.

The uniprocessor abstract machine has a single thread of execution. At each step, this thread takes the first program at the head of the queue and acts upon it. In the example above, it takes the program $u(y).P$ which wishes to perform input on channel u. Since there is already another program waiting to output on channel u, the thread of execution allows them to react, and places the remainders $P\{x/y\}$ and S back in the queue:



Queue: $\overline{u}z.Q;\ R;\ P\{x/y\};\ S$

The next head of the queue $\overline{u}z.Q$ wishes to perform an output on channel u. But there is no other program waiting to input. Therefore, the program is placed in the channel, thereby marking it as 'waiting':



Queue: $R;\ P\{x/y\};\ S$

There are other transition steps that act only upon the queue, not channels. For instance, $P|Q; R \longrightarrow R; P; Q$ interprets parallel composition in the pi calculus, and $\mathbf{0}; R \longrightarrow R$ interprets the nil process. Also, if the program at the

head of the queue is a piece of code in a functional or imperative language, then it can be executed in its own way.

Cardelli describes a modification to the machine so it can implement the summation operator: a program may be waiting simultaneously in several channels, but as soon as its first 'wait' is liberated, all the others are aborted. Pierce and Turner describe a variation of the machine which fulfills the *fairness* property that a waiting program will not be starved of interaction. To this end they use a queue of waiting programs for each channel, rather than an unsorted set. One consequence of this is that the machine is deterministic, and hence not an exact match for the (inherently non-deterministic) pi calculus. They also considers optimisations such as the use of an environment rather than substitution: environments are faster in a uniprocessor implementation. However, in a distributed implementation, it may prove too costly to transport environments. Optimisation is important for Pierce and Turner because their project is an implementation for the pure pi calculus, without fragments of other languages: therefore synchronous rendezvous is the only mechanism left for performing actual computation, and so it happens frequently. A modification by Sewell and Wojciechowski [71] allows a new thread of execution to be spawned, in the case that the program at the front of the queue contains a blocking operating system call such as 'wait until the user presses a key'.

The key questions for a distributed implementation are: Where in the system should the queue reside? Should there be one queue, or many? Where should the channels reside? How can we perform a distributed substitution? In the following sections I give my answers.

1.4 Channel-managers and pre-deployment

In distributed calculi it is common to talk about *channel-managers*. These are devices that exist somewhere in the system, and that act as mediators for rendezvous. One program sends a message to the channel-manager requesting to receive data; another program sends a message to the channel-manager requesting to send data; and the channel-manager pairs them up and sends confirmation messages back.

Often, programs in distributed calculi are viewed as unitary entities with prolonged existence, called *agents*. A system might therefore contains both agents and channel-managers, and have messages sent between the two. (In distributed calculi the agents are also usually able to move themselves about. However, although such mobile agents are widely studied theoretically, they have so far been used rarely in practice.)

Some distributed calculi use only agents and no channel-managers: all messages are sent between agents, and each agent has a list of channels on which it can accept messages. Other calculi combine agents with channel-managers: for each channel there is only a single agent in the system which can receive messages on it.

In this section I introduce a different model in which agents do not exist at runtime. The only entities which exist are channel-managers; all programs are fragmented between them. I explain with four examples.

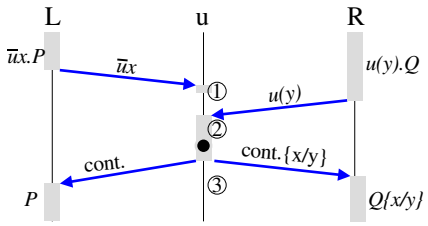


Figure 1: Facile interaction between a program $L = \overline{u}x.P$ on the left and $R = u(y).Q$ on the right, interacting through a channel-manager u in the middle. This is an *interaction diagram* [7]: a vertical line represents a program waiting for messages; a shaded box represents a program executing; diagonal lines represent messages, and are annotated with the content of the message. At time (1), the channel-manager contains a single output command. At time (2), it contains both input and output and so allows the reaction (indicated by the black circle). At time (3), the channel-manager is empty again.

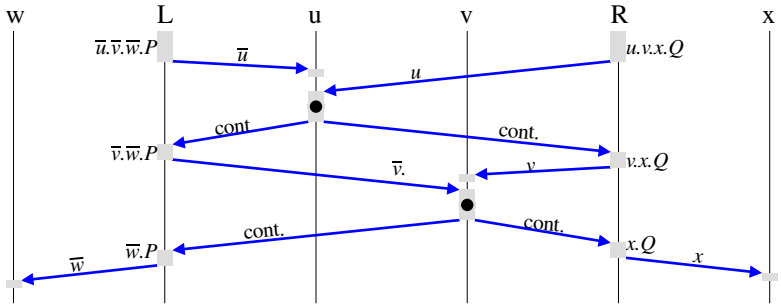


Figure 2: Several Facile interactions in sequence for $L = \overline{u}.\overline{v}.\overline{w}.P$ in parallel with $R = u.v.x.Q$. Each interaction involves a four-message three-party 'handshake': two messages to propose the communication to the channel-manager, and two continuation messages sent back to L and R. Note that the agents L and R, and the channel-manager u, are entities that exist somewhere in the system, and so each has its own vertical line.

Note that, in the agent model, one usually annotates in the calculus whether $P|Q$ denotes two agents, or one agent containing a program with two concurrent threads of execution. In my model I leave $P|Q$ un-annotated in the calculus, and then specify at the implementation-level which fragments of P and Q are placed at which channel-manager.

The first example is Facile [26, 38], a language and its implementation which integrate the pi calculus with the lambda calculus. In Facile, messages are exchanged between agents and channel-managers—see Figures 1 and 2. But it is unappealing as a distributed implementation of the pi calculus because of its handshaking: if one of the 'continue' messages were to be lost, the resulting intermediate state would not be expressible in the calculus. The handshaking also has a high *latency*: first one message must be sent, and then another must be received, and these two tasks must be done sequentially rather than concurrently.

The second example is from the join calculus [17]. See Figure 3. The join calculus combines agents with channel-managers: every entity in the system is

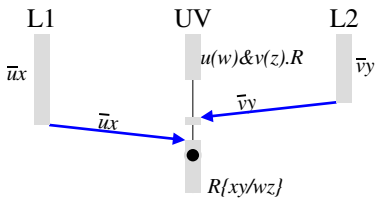


Figure 3: Join interaction with programs $L1 = \overline{u}x$, $L2 = \overline{v}y$ and channel-manager $UV = u(w)\&v(z).R$. Synchronisation is achieved through a *join pattern* at the received, in this case $u(w)\&v(z)$, which waits until it has received from both senders. Then the program R might go on to send asynchronous messages to other channel-managers.

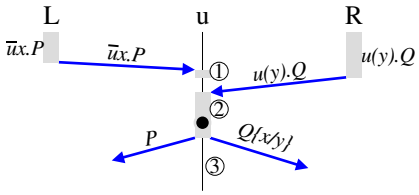


Figure 4: Continuation machine showing interaction between $L = \overline{u}x.P$ and $R = u(y).Q$. Each message includes an entire program. At time (1), the channel-manager u contains just the program $\overline{u}x.P$. At time (2), it contains $\overline{u}x.P|u(y).Q$, and allows a reaction. At time (3) it contains nothing.

a channel-manager which is ready to accept messages. In fact, it waits to accept combinations of messages rather than individual messages. Once it has received such a combination, it might send further messages to further channel-managers. The command to send messages is asynchronous (i.e. no continuation can follow it); this means that there is no handshaking.

Like join, many calculi limit themselves to asynchronous-send, with the intention of eliminating handshakes. It is possible to encode synchronous rendezvous with asynchronous messages; indeed, this is how Facile is implemented. However, such an encoding suffers from the same handshaking problems as Facile. I believe that synchronous rendezvous is such a natural way to write programs that a handshake-free implementation would bring real benefit.

With the following example, I introduce a handshake-free implementation of synchronous rendezvous. I call this example the *continuation machine*. See Figures 4 and 5. There are no agents. A program such as $\overline{u}.P|v.Q$ is split up into its parallel components $\overline{u}.P$ and $v.Q$. The first component is then deployed in its entirety to channel u, and the second to channel v. Once a component has reacted, it then liberates further components which themselves are deployed to channels. Note, incidentally, that the uniprocessor implementation of the pi calculus (discussed in the previous section) also places parallel components of a program in different channel-managers.

The continuation machine has two advantages. First, it uses half as many messages as Facile. Second, it manages—without limiting itself to asynchronous rendezvous—to eliminate handshaking. This reduces latency compared to Facile and ensures that, even if a message should be lost, the result is still expressible in the pi calculus.

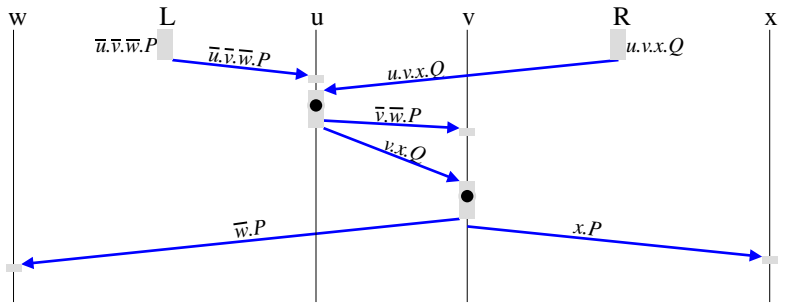


Figure 5: Several continuation interactions in sequence for $L = \overline{u}.\overline{v}.\overline{w}.P$ in parallel with $R = u.v.x.Q$. Observe that, after reaction at u, no messages need be sent back to L or R: execution can instead continue directly at v. L and R are not agents; they merely indicate the starting location of the programs. Once the programs have been deployed, there is no further use for L and R.

However, the continuation machine has a serious problem with total message volume. Consider how it executes the program $\overline{u}.\overline{v}.\overline{w}.\overline{x}.\overline{y}.\overline{z}$. First this program is deployed in its entirety to channel u. Next, $\overline{v}.\overline{w}.\overline{x}.\overline{y}.\overline{z}$ is deployed to v, and so on. For a sequence of length n, there will be a total of n messages, but the total volume of all these messages will be $\frac{1}{2}n^2$. Consider also the example $\overline{u}.\overline{v}.\overline{w}.P$ where P is some program, perhaps in an imperative language. The continuation machine will transport this P three times around the network. This is not acceptable on a slow network, nor if P is large.

To avoid the repeated cost of transporting P, my solution is to deploy the fragment P to its final destination w right at the start. In the final example I introduce a machine which deploys fragments in this way and still achieves handshake-free operation. I call this machine the *deployment machine*. See Figures 6. Again, there are no agents. A program $\overline{u}.\overline{v}.\overline{w}.P \mid u.v.x.Q$ might be divided up into fragments $\overline{u}$, $\overline{v}$, $\overline{w}.P$, u, v and $x.P$ which are then all deployed. Each fragment in the machine is marked as blocked or unblocked; in this example, $\overline{u}$ and u are the only unblocked fragments. When two unblocked fragments are at the same channel, they react together. They then send messages to unblock their subsequent fragments.

Let us consider how our example program $\overline{u}.\overline{v}.\overline{w}.\overline{x}.\overline{y}.\overline{z}$ will execute on the fusion machine. First, each fragment is deployed to its appropriate channel. Given that there are n commands, deployment takes n messages each of a constant size. As the program executes it takes a further n messages, just as in the continuation machine, and each of these is a constant-sized 'continue' message. In summary, this program takes $2n$ messages with total size $2n$ in the deployment machine, whereas in the continuation machine it takes n messages with total size $\frac{1}{2}n^2$.

Fragmentation is a good implementation principle for three reasons. Unlike Join, it allows for synchronous rendezvous. Unlike Facile, it avoids handshaking. And unlike the continuation machine, the total volume of all of messages remains small.

The idea of fragmentation may be unappealing to some readers, especially those familiar with mobile agents. However, by a twist of perspective, many existing systems can be thought of as using fragments. Consider for instance

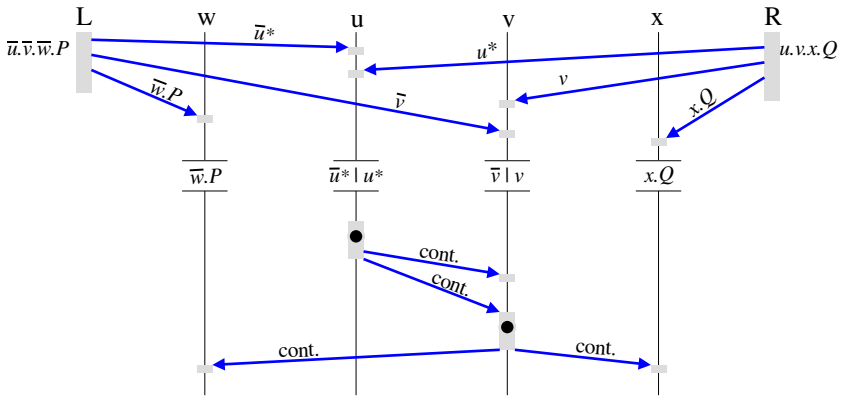


Figure 6: Deployment machine executing the programs $L = \overline{u}.\overline{v}.\overline{w}.P$, $R = u.v.x.Q$. First, the programs are divided up into fragments, and each fragment is deployed to its desired destination. The fragments $\overline{u}$ and u are marked with an asterisk to indicate that they are 'top-level' and can executed immediately; the other fragments are blocked. (The diagram indicates the contents of each channel within two horizontal lines). The program proceeds to execute. Each continuation message unblocks a further fragment.

the hyper-text transfer protocol http, the mechanism by which web pages are transmitted. One might normally say that the protocol describes two separate programs, one running on the web-server and the other running on a client's machine. But we might instead view the protocol as describing a single program, with one fragment deployed on the web-server and the other fragment deployed on the client's machine.

This explanation of the deployment machine leaves two key problems unaddressed: how to effect a substitution whose scope spans several fragments, and how to encode blocking and continuation messages. The following sections outline how both problems can be solved with explicit fusions.

The *fusion machine* presented in Chapters 5 and 6 is actually a continuation machine. It is able to execute programs in both the explicit fusion calculus and the pi calculus, without needing to translate one into the other. However, we will see that fragmentation can be encoded purely within the explicit fusion calculus. When the continuation machine executes such an encoded program, it behaves just like a deployment machine. Our encoding therefore provides a more efficient way to execute programs in the pi calculus as well as the explicit fusion calculus.

1.5 Fragmentation

The previous section has shown how fragmentation gives efficiency. We now consider how to express fragmentation within a pi-like calculus.

The following example shows a simple fragmentation. The program on the left is the original; the program on the right has been fragmented, perhaps because P is a large piece of code which is too costly to move about repeatedly:

$$u.v.w.P \qquad\qquad (t)(\, u.v.w.\overline{t} \mid t.P \,)$$

The un-fragmented program waits for an (empty) rendezvous on u, then v, then w, then continues with P. We say that P is *syntactically guarded* by w, since the syntax itself directly indicates that P cannot execute until after w. Now consider the fragmented program to the right. It uses a private (local) channel named t for 'trigger'. The fragment $t.P$ runs concurrently with the rest of the program, but it is idle until the rest of the program outputs on t. Here we say that P is *semantically guarded* by w: the fact that it cannot execute until after w is a property of the behaviour of the program. Note that fragmentation is a transformation applied to one program in the calculus, which yields another program also in the calculus.

More generally, we need to fragment programs with non-empty communication. Consider the following example:

$$u(x).v(y).w(z).P \qquad\qquad (t)(\, u(x).v(y).w(z).\bar{t}xyz \mid t(xyz).P\,)$$

In the program on the left, the names x, y, z are received through communication. These are local names whose scope includes the continuation P. When we fragment the program (on the right), the scope of the names no longer includes P: we must therefore send it copies of these names. Effectively, we must transport an entire environment at every fragment boundary. In a large program with lots of fragments and large environments, this is too costly.

The only way to avoid the cost of copying names is for the same names x, y and z to be used in both fragments, rather than having private copies used in P. If the same names are to be used in both fragments, then the first fragment cannot use binding ('scope-limiting') input. This leads to the following fragmentation:

$$u(x).v(y).w(z).P \qquad\qquad (txyz)(\, ux.vy.wz.\bar{t} \mid t.P\,)$$

A form of fragmenting first appeared as part of the *solos* calculus of Laneve and Victor [34]. This is a version of the pi calculus in which both input and output lack continuations; it is therefore necessarily completely fragmented. The surprising result is that the solos calculus is computationally as powerful as the pi calculus, even though it cannot express trigger guards directly. Laneve and Victor demonstrated this with an encoding of the prefix operator.

In fact, fragmentation does not require that we do away with all continuations, as shown in the examples above. In Chapter 6, I present an encoding of the prefix operator that uses limited (non-nested) continuations. I prove that this is more efficient than Laneve and Victor's encoding.

1.6 Forwarders and fusions

We now consider how to implement fragmentation. Our motivating example is the following program, shown both in original and fragmented form.

$$w(x).P.Q \qquad\qquad (xtu)(\, wx.\bar{t} \mid t.P.\overline{u} \mid u.Q\,).$$

When the program reacts, it gives rise to a substitution. The substitution affects both P and Q:

$$\overline{w}y \mid (xtu)(\, wx.\bar{t} \mid t.P.\overline{u} \mid u.Q\,) \quad\longrightarrow\quad (tu)(\, \bar{t} \mid t.P\{y/x\}.\overline{u} \mid u.Q\{y/x\}\,).$$

It will take two separate messages to notify both fragments about the substitution, since they are at separate locations. But if one of these messages were to be lost, the result would be one substituted fragment and one unsubstituted fragment. This amounts to a reaction step that is not expressible in the calculus. Therefore, the calculus should not use such a large step for reaction: it should use smaller atomic steps. We will see how explicit fusions allow for these smaller steps. But first we consider related work: in particular, substitutions and forwarders.

An explicit *substitution* (not fusion) is a substitution written explicitly in a term. It has duration, and with it one can delay or control the substitutive effect of reaction. This allows for reaction to be atomic. Explicit substitutions were first used in the lambda calculus [1]. The fusion machine also uses a version of explicit substitutions in its implementation, although for practical reasons explained below we find it more natural to use explicit fusions at the calculus level. More recently, a substitutive explicit term called an *active substitution* has been introduced for the pi calculus [2], as a theoretical tool for bisimulation proofs.

Forwarders are a practical incarnation of explicit substitutions. A forwarder is a device which, when it receives a message or program fragment, sends it on to somewhere else. Wojciechowski [71] shows how to encode a forwarder in Nomadic Pict, and reviews common optimisations. A recent distributed implementation of the ambient calculus [19] uses forwarders to help avoid handshakes. To see how forwarders help avoid handshakes, consider the following example. Imagine that a mobile phone is moving from Cambridge to Bologna and that, during the move, all messages that arrive in Cambridge get forwarded to Bologna. Then someone across the world in Australia can send messages to either location, interchangeably, without needing to handshake with the phone to send directly to its current location.

We will consider how forwarders can also be used to implement small-step reaction. Let $x \rightsquigarrow y$ stand for a forwarder from x to y, and suppose that there is a fragment $x.P$ located at x. Then $x.P \mid x \rightsquigarrow y$ will evolve to $y.P \mid x \rightsquigarrow y$. We can now express reaction in small atomic steps, by supposing that it merely gives rise to a forwarder:

$$(x)(\,\overline{w}y \mid wx \mid x.P\,) \quad \longrightarrow \quad (x)(\,x \rightsquigarrow y \mid x.P\,).$$

Subsequent atomic steps with the forwarder will transform $x.P$ into $x.P\{y/x\}$.

However, a problem arises when forwarders conflict. Consider the example

$$\overline{w}z.\overline{u}y \mid wx.ux \quad \longrightarrow \quad x \rightsquigarrow z \mid \overline{u}y \mid ux \quad \longrightarrow \quad x \rightsquigarrow z \mid x \rightsquigarrow y.$$

The result has two forwarders leading out from x. Therefore, two messages sent to x might fail to meet. These conflicting forwarders must be resolved in some way, perhaps by turning them into $x \rightsquigarrow z \mid z \rightsquigarrow y$. The end result is that the rest of the program can refer to x or y or z, interchangeably. That is to say: given non-binding input, and given that there are rules for conflict resolution, reaction in the calculus gives rise to an equivalence on names—a fusion.

We use forwarders in the fusion machine, and give rules for conflict resolution (Chapter 5). However, the rules are subtle and require substantial proof. It also turns out in the bisimulation theory (Chapter 3) that the ability to interchange backwards and forwards between names, rather than just substitute in one direction, is frequently useful.

Because conflict-resolution is awkward, and because interchangeability is natural, we use explicit fusions in the calculus. An explicit fusion is a term $x{=}y$ that allows a reversible interchange between names: $x{=}y \mid x.P \equiv x{=}y \mid y.P$. This amounts to abstracting away the rules for conflict-resolution. The previous example now becomes

$$\overline{w}z.\overline{u}y \mid wx.ux \quad \longrightarrow \quad x{=}z \mid \overline{u}y \mid ux \quad \longrightarrow \quad x{=}z \mid x{=}y.$$

Conflict is by nature impossible with explicit fusions: even if one fragment $\overline{x}.P$ has been turned into $\overline{z}.P$, and another $x.Q$ has been turned into $y.Q$, the two fragments can still be reunited by reverting back to x.

We have discussed the problem of conflicting forwarders, and how we solved it with conflict resolution. But a second solution is possible: we could avoid all conflict in the first place. The obvious way to do this is to mandate that every received name is received exactly once (like bound input in the pi calculus), and then create forwarders with the rule $\overline{u}y.P \mid u(x).Q \longrightarrow x \rightsquigarrow y \mid P \mid Q$. Because x is bound, this same x will never be used in input elsewhere, so we will never produce a conflicting $x \rightsquigarrow z$. This technique is used by Abadi and Fournet for their active substitutions, for the following technical reasons. Their goal is to allow not just names but also fragments of code to replace received names, so that for instance $x \rightsquigarrow P_1 \mid x.Q$ evolves to $P_1.Q$. But therefore they cannot resolve $x \rightsquigarrow P_1 \mid x \rightsquigarrow P_2$, since it is meaningless in $x \rightsquigarrow P_1 \mid P_1 \rightsquigarrow P_2$ to forward from one piece of code to another.

We see that bound input guarantees that there will be no conflict. However, bound input cannot express fragmentation. If we are to limit ourselves to a calculus with bound input, and still want fragmentation for efficiency reasons, then we would need to invent some sub-calculus which allows fragmentation, and then fragment from the pi calculus into this sub-calculus. The sub-calculus would have to use non-binding input, as explained in the previous section. It would therefore be very similar to the explicit fusion calculus.

We therefore face a choice: either implement the explicit fusion calculus and provide rules for conflict-resolution; or implement the pi calculus without rules for conflict-resolution and prove that conflict never arises. But it gains neither efficiency nor parsimony to omit the rule for conflict resolution: When executing a pi calculus program, conflict never arises, so the rule for conflict resolution causes no inefficiency. Even if the rule is omitted, an implementation still needs some rule to create forwarders, but the fusion machine combines both conflict-resolution and forwarder-creation in a single simple rule.

On the other hand, we gain flexibility and symmetry by including the rule for conflict resolution. We are able to implement the full language we use for fragmentation, rather than just a subset of it. And we can implement the fusion calculus and the solos calculus.

Because conflict-resolution brings benefit at no cost, it should be retained: effectively, it makes sense to implement the explicit fusion calculus.

1.7 Whether explicit locations are needed

It is commonly felt that a calculus for distribution ought to have some way to mark co-location of programs—a notion absent from the pi calculus. In a distributed system, messages between co-located programs are fast, while

messages between remote (not co-located) programs are slow and might get lost. And often, when one program fails, it is because of a crash that affects all other co-located programs. If we mark co-location in our programs, then we can reason about their failure-safety and efficiency.

The typical approach is to add a class of entities called *locations* to the calculus, with each location containing a number of programs. New rules of interaction are added which operate upon locations. This approach is used, for instance, by the distributed pi calculus [57], the pi-l calculus [5], the nomadic pi calculus [64] and the ambient calculus [10].

I propose a different approach. I do not add any new entities to the calculus. Instead, I consider an equivalence-relation on channel names, with the intended meaning that related names indicate co-located channels. This equivalence relation then modifies the operational semantics of the calculus: messages between co-located channels are efficient, and messages between remote channels may be lost. (In fact, in this dissertation, I consider only efficiency. Locations in distributed calculi have been widely to address failure, but I am not aware of any previous work on efficiency.) My approach might be described as adding just *co-location*, without actually adding location.

The technical advantage of co-location is that it keeps the calculus simple. We can even ignore co-location entirely when reasoning about correctness. We can also use more detailed metrics for the distance between channels, perhaps to indicate signal strength in a wireless network, without needing to alter the calculus itself.

Co-location in this sense has not been used before. The closest similarity is with current work by Milner [46], in which he characterises programs by two graphs: one for its possible communications, and one for its co-location. He explains that it is easier to prove properties about communication, because he can ignore the additional complexity of co-location.

The question of whether to add location or co-location is open to debate. It partly depends on what we are trying to model. When modelling the Internet, one might say that locations correspond to Internet Protocol (IP) numbers. The Internet fabric is able to map an IP number to a particular machine (although various new technologies alter this mapping), and it is able to deliver messages to IP numbers. We might add location-based commands to our calculus to model these IP messages, and retain the pi calculus only for communication within a machine.

On the other hand, when programming applications that use the Internet, it is usual to address messages not only by a machine's IP number but also by a *port* number within that machine. Exactly the same commands are used to send a message to a program on the same machine but different port, as to send to a program on a different machine. (The message will be delivered more quickly and reliably in the first case.) We might therefore take each channel name to represent a combination of IP number and port number—basically a *socket* [66]. Because the commands in our language only use channel names, we can make do with adding co-location rather than location.

Conclusions. The preceding four sections have described the key features of the fusion machine:

- at runtime, the only things that exist are channels;

- programs are fragmented between channels;
- forwarders are used to simulate explicit fusions;
- and rather than location, the machine uses co-location.

The fusion machine itself is described in Chapters 5 and 6. It builds heavily upon the explicit fusion calculus, which is described in Chapters 2 to 4.

Chapter 2

Explicit fusions

An explicit fusion is a term in a calculus which allows two names to be used interchangeably. In this chapter, I present the *explicit fusion calculus*—a variant of the pi calculus with explicit fusions. The first section describes it by example, the second and third sections define it, and the fourth briefly outlines some key connections with related work.

In the pi calculus, the word *process* has come to mean simply a term in the calculus; and when two processes are placed in parallel, they share names and the result is still called a process. This word is unfortunate for programmers. An *operating system process* is something with its own protected address space, so that two processes in parallel do not share names. Each operating system process may contain one or more *threads of execution*. To avoid confusion, I refer to them as *terms* or *programs* in the pi calculus, rather than processes.

2.1 Explicit fusion calculus

This section describes the explicit fusion calculus. I assume that the reader is familiar with the pi calculus. In the pi calculus, reaction between input and output happens in a single step:

$$\overline{u}x.P \mid u(y).Q \;\longrightarrow\; P \mid Q\{x/y\}$$

But we will use explicit fusions to analyse the reaction in smaller steps. *At the instant of interaction, let us say that the name y and the local name x become fused:*

$$\overline{u}x.P \mid u(y).Q \;\longrightarrow\; (y)(\, x{=}y \mid P \mid Q\,) \tag{1}$$

This means that P and Q can refer to x or y, interchangeably. Since the two names are interchangeable, we can substitute x for y throughout Q. Finally, since the local name y is merely an alias for x, and since it is not being used, we can dispense with it:

$$(y)(\, x{=}y \mid P \mid Q\,) \;\equiv\; (y)(\, x{=}y \mid P \mid Q\{x/y\}\,) \;\equiv\; P \mid Q\{x/y\}$$

I have chosen to express the substitutive effect of explicit fusions through the structural congruence $\equiv$ rather than reaction $\longrightarrow$. There are two reasons for

this. First, it means that a single reaction in the explicit fusion calculus corresponds to a single reaction in the pi calculus. Second, it takes extra work to implement fusions in a directed way, and reaction is directed: it is more convenient to postpone that extra work for now, by sticking to (undirected) structural congruence. We will study a directed implementation of fusions in Chapter 5, as part of the fusion machine.

The explicit fusion calculus uses non-binding input. When non-binding input reacts with non-binding output, the result is an explicit fusion:

$$\overline{u}x.P \mid uy.Q \;\longrightarrow\; x{=}y \mid P \mid Q$$

This reaction is a local one between the output and input commands. But the effect of the resulting fusion is global in scope: x and y can be used interchangeably throughout the entire term, even in other parallel terms. To limit the scope of the fusion we use restriction. Here we rewrite (1) to use non-binding input:

$$\overline{u}x.P \mid (y)(uy.Q) \;\equiv\; (y)(\,\overline{u}x.P \mid uy.Q\,) \;\longrightarrow\; (y)(\,x{=}y \mid P \mid Q\,)$$

2.2 Explicit fusion calculus, formally

We now define the explicit fusion calculus. Let there be a set $\mathcal{N}$ of names ranged over by $u, v, \ldots$. We write $\tilde{u}$ for a possibly-empty finite sequence $u_1, \ldots, u_n$ of names. Let there also be a set of *co-names* $\overline{\mathcal{N}} = \{\overline{u} : u \in \mathcal{N}\}$. Let μ range over $\mathcal{N} \cup \overline{\mathcal{N}}$, and write $\overline{\mu}$ to interchange μ between name and co-name.

Definition 1 (Explicit fusion calculus) *The set $\mathcal{P}_\phi$ of explicit fusion terms is given by*

$$P \quad ::= \quad \boldsymbol{0} \;\mid\; P|P \;\mid\; !P \;\mid\; (x)P \;\mid\; \overline{u}\tilde{x}.P \;\mid\; u\tilde{x}.P \;\mid\; x{=}y$$

We define bound and free names in the standard way: the restriction operator $(x)P$ binds x in P. The name x is free if it occurs in the *subject* μ of an action $\mu\tilde{x}.P$, either as name or co-name, or the *object* $\tilde{x}$ of that action. It is also free in the fusions $x{=}y$ and $y{=}x$. We write $\mathrm{fn}(P)$ to denote the set of free names in P. We use the abbreviations $(\tilde{x})P = (x_1)\ldots(x_n)P$ and $\tilde{x}{=}\tilde{y} = x_1{=}y_1 \mid \ldots \mid x_n{=}y_n$. We sometimes write ϕ for a fusion $\tilde{x}{=}\tilde{y}$ when it is not important which names are being fused.

In the symbol $\mathcal{P}_\phi$, the subscript ϕ stands for 'explicit fusion' terms. Later we will define $\mathcal{P}_\pi$ for pi calculus terms.

Definition 2 (Contexts) *The set $\mathcal{E}_\phi$ of explicit fusion contexts is given by*

$$E \quad ::= \quad \_ \;\mid\; \mu\tilde{x}.E \;\mid\; !E \;\mid\; (x)E \;\mid\; P|E \;\mid\; E|P$$

Definition 3 (Structural congruence) *The structural congruence $\equiv$ between terms is the smallest equivalence relation satisfying the following axioms, and closed with respect to the contexts $\_ \mid \_$, $!\_$, $(\_)\_$ and $\mu\tilde{x}.\_$:*

1. *Abelian monoid laws with $\boldsymbol{0}$ as identity*
   $P \mid \boldsymbol{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

2. *scope laws*
   $(xy)P \equiv (yx)P \qquad (x)(P \mid Q) \equiv (x)P \mid Q$ *if* $x \notin \mathrm{fn}(Q)$

3. *replication*
   $!P \equiv P \mid !P$

4. *equivalence laws of fusion*
   $x{=}x \equiv \mathbf{0} \qquad x{=}y \equiv y{=}x \qquad x{=}y \mid y{=}z \equiv x{=}z \mid y{=}z \qquad (x)(x{=}y) \equiv \mathbf{0}$

5. *interchange law of fusion*
   $x{=}y \mid P \equiv x{=}y \mid P\{y/x\}$

The first three lines are standard from the pi calculus. The fourth line gives the equivalence properties of explicit fusions: they generate equivalence relations on names, parallel composition is the equivalence-closed union of two such relations, and restriction removes a name from a relation. We define this equivalence relation in the following section. Together, the final two lines allow us to derive alpha renaming. For example:

$$\begin{aligned}
(x)(\overline{x}.\mathbf{0}) &\equiv (xy)(x{=}y \mid \overline{x}.\mathbf{0}) && \textit{create fresh bound name } y \textit{ as an alias for } x \\
&\equiv (xy)(x{=}y \mid \overline{y}.\mathbf{0}) && \textit{interchange } x \textit{ for } y \\
&\equiv (y)(\overline{y}.\mathbf{0}) && \textit{remove the now-unused bound name } x
\end{aligned}$$

Strictly speaking, rule 5 involves a capture-avoiding substitution, which implicitly requires alpha renaming. This means that, although alpha renaming is not needed as a rule in the structural congruence, it is still used as a sub-definition. To completely remove the need for a capture-avoiding substitution, we could replace rule 5 with a substitution rule that only affects the subject and object of an action, and add $x{=}y \mid \mu\tilde{x}.P \equiv x{=}y \mid \mu\tilde{x}.(x{=}y \mid P)$. Gardner and I develop this idea in [25].

Definition 4 (Reaction) *The reaction relation $\longrightarrow$ is the smallest relation satisfying the following rule, and closed with respect to structural congruence and to the contexts $\_ \mid \_$ and $(\_)\_$.*

$$\overline{u}\tilde{x}.P \mid u\tilde{y}.Q \;\longrightarrow\; \tilde{x}{=}\tilde{y} \mid P \mid Q.$$

Note that prefix is a non-reactive context. That is, in the term $\mu.P$, the term P cannot react until after the μ. In this respect, prefixing is like the sequencing operator (semicolon) in imperative programming languages. Prefixes are also known as *guards*.

2.3 Equivalence relation

Through the structural congruence, the explicit fusions in a term generate an equivalence relation on names. We define it here because it plays such a pervasive part in the explicit fusion calculus. More generally, any concurrent calculus that has explicit fusions, parallel composition and restriction generates an equivalence relation:

- *Explicit fusions* provide a finite basis for an equivalence relation on names.
- *Parallel composition* of two terms in the calculus generates the transitive closure of the terms' equivalence relations.

- *Restricting* a name in a term in the calculus generates the term's equivalence class with that name removed.

Strictly speaking, these three points merely indicate how *an* equivalence relation can be generated from the syntactical structure of some term in a calculus. For each calculus, it is necessary to prove that this equivalence relation is the 'correct' one. This section gives that proof for the explicit fusion calculus. That is to say: through the operations of structural congruence, the explicit fusions in a term give rise to an equivalence relation, and we prove that this relation is equal to the relation we defined structurally.

Definition 5 (Equivalence relation) *An equivalence relation is a binary relation that is closed with respect to transitivity, symmetry and reflexivity. Let E, F be equivalence relations on a set $\mathcal{N}$, and $x, y, z \in \mathcal{N}$. Define*

$$\begin{aligned} E \oplus F &= \{(x,y) : \exists \tilde{z} : (x,z_1), \ldots (z_n,y) \in (E \cup F)\} && \textit{equivalence-closed union} \\ E\backslash z &= \{(x,y) : (x,y) \in E \wedge z \notin \{x,y\}\} \cup \{(z,z)\} && \textit{removal of an element} \end{aligned}$$

The identity relation $\mathbf{I}$ is an equivalence relation.

We now prove that the specified equivalence relation is indeed generated by the structural congruence.

Definition 6 *A term P in the explicit fusion calculus generates an equivalence relation* $\mathrm{Eq}(P)$ *on names as follows.*

$$\begin{aligned} \mathrm{Eq}(x{=}y) &= \{(x,y),(y,x)\} \cup \boldsymbol{I} && \textit{smallest equivalence relating } x \textit{ to } y \\ \mathrm{Eq}(P|Q) &= \mathrm{Eq}(P) \oplus \mathrm{Eq}(Q) && \textit{equivalence-closed union} \\ \mathrm{Eq}((x)P) &= \mathrm{Eq}(P)\backslash x && \textit{removal of an element} \\ \mathrm{Eq}(!P) &= \mathrm{Eq}(P) \\ \mathrm{Eq}(\mu\tilde{x}.P) &= \boldsymbol{I} \\ \mathrm{Eq}(\boldsymbol{0}) &= \boldsymbol{I} \end{aligned}$$

We write $P \vdash x{=}y$ as shorthand for $(x,y) \in \mathrm{Eq}(P)$.

Lemma 7 *$P \equiv Q$ implies $\mathrm{Eq}(P) = \mathrm{Eq}(Q)$*

Proof. By rule induction on the derivation for $P \equiv Q$ (Definition 3). For the rule for fusion interchange, we use the result that $\mathrm{Eq}(()P\{y/x\}) = \big(\mathrm{Eq}(()P) \oplus \mathrm{Eq}(()y{=}x)\big)\backslash x$, proved by induction on the structure of P. □

Corollary 8 *$P \vdash x{=}y$ if and only if $P \equiv x{=}y \mid P$*

2.4 Work related to fusions

This section outlines some key connections between the explicit fusion calculus and related work. All connections are dealt with more formally in subsequent chapters.

Consider the following example of a pi calculus program $\overline{u}x.P \mid u(y).Q$, translated into the explicit fusion calculus and executing according to the laws

of the explicit fusion calculus:

$$\begin{array}{rcll} \overline{u}x.P \mid (y)(uy.Q) & \equiv & (y)(\overline{u}x.P \mid uy.Q) & \textit{scope extrusion} \\ & \longrightarrow & (y)(x{=}y \mid P \mid Q) & \textit{reaction} \\ & \equiv & (y)(x{=}y \mid P \mid Q\{y/x\}) & \textit{fusion interchange} \\ & \equiv & P \mid Q\{y/x\} & \textit{remove now-unused name } y \end{array}$$

Note that whenever the left hand side of a reaction is a translated pi calculus term, then all explicit fusions can be dismissed from the result, also yielding a pi calculus term. We say that *piability* is preserved. In Section 4.5 (page 69) we define piability, and prove that a pi term can react if and only if its translation can also react. We also prove that the translation is sound with respect to bisimulation congruence.

More generally, we can get rid of all explicit fusions in the result if and only if the left hand side has no explicit fusions and satisfies a particular side condition. The side condition is that at least one of the names x and y is restricted. For instance:

$$\begin{array}{rcll} (x)(\overline{u}x \mid uy \mid P) & \longrightarrow & (x)(x{=}y \mid P) \equiv P\{y/x\} & \textit{Can remove fusion} \\ \overline{u}x \mid uy \mid P & \longrightarrow & x{=}y \mid P & \textit{Cannot remove fusion} \end{array}$$

The fusion calculus [70] and the chi calculus [21] have this side condition for their reaction, that one of the names is restricted. They need it because they have non-binding input but lack explicit fusions, and so cannot allow any explicit fusions to remain in the result. It is hard to generalise the condition to polyadic reaction without mentioning explicit fusions. We will recall how it is done in the fusion calculus (Definition 46, page 62). We will also prove that a term can react in the fusion calculus if and only if it satisfies the side condition and can react in the explicit fusion calculus. And we prove that the explicit fusion calculus is a fully abstract model, with respect to bisimulation congruence, for the fusion calculus. Explicit fusions provide a simpler account of the fusion calculus.

Let us compare the pi calculus and the fusion calculus. Both essentially require that, after reaction, all explicit fusions can be dismissed. The pi calculus achieves this end with a constraint on admissible terms (it only allows bound input). The fusion calculus instead achieves it with a constraint on admissible reactions.

There is a surprising result about the fusion calculus due to Laneve and Victor [34]: a term with guards can be translated into one without, while preserving (weak barbed) bisimulation. The fusion calculus without guards is called the *solos calculus*. Fu [22] has also independently studied solos in the context of proof nets. The translation from the fusion calculus into the solos calculus requires non-binding input. It uses non-binding input in its *catalyst agents* $U_y = (z)yzz$, which listen on a channel y and then fuse the two names that come. For example, the guarded term $u.\overline{v} \mid \overline{u}$ would be translated into

$$(xy)\big(\, U_y \mid \overline{u}y \mid uz \mid \overline{z}vx \mid \overline{x} \,\big). \qquad \textit{Solos calculus}$$

The reaction of this term is as follows. First the two channels on u can react immediately. This fuses z to the catalyst at y. The catalyst then reacts on z to fuse x with v, yielding the desired result. Note that this encoding uses an extra

reaction step and carries extra data over the channel u: it is therefore not a congruence, and only works up to weak bisimulation. Less obviously, this encoding cannot be distributed in the fusion machine without costing extra inter-location messages. In Section 6.8 (page 113), I exhibit a different encoding which allows fusion guards $\overline{u}.(x{=}y)$ but no other guards. I prove that my encoding is a congruence, and works up to strong bisimulation, and can be distributed without costing extra messages. Note that the example above is a simplification of the catalyst encoding: the full version is given in Section 6.10 (page 118).

An *equator* is a program that implements a fusion but is written in the pi calculus, using the standard output and input commands of the pi calculus. Equators were introduced by Honda and Yoshida [31], as part of a project to characterise the equivalence of two programs based solely on the internal reactions they can perform. Equators are defined as follows:

$$\mathcal{E}(u,v) \stackrel{\text{def}}{=} !u(x).\overline{v}x \mid !v(x).\overline{u}x. \qquad \textit{Equator}$$

Essentially, one part of the equator forwards messages from u to v, and the other part 'backwards' messages. An equator thereby allows the two channel names to be used interchangeably. However, equators are more awkward than explicit fusions. While explicit fusions generate an equivalence relation which we have completely characterised, equators generate their equivalence relation only through weak bisimulation, and so a complete characterisation is not possible.

Equators are also awkward with respect to program equivalence. Suppose that two programs are equivalent, and we observe that one can accept a message over a channel. We normally expect that the other program should also be able to accept a message over that channel. But an equator is 'always on', always able to receive a message. We no longer gain information (discriminating power) through observing that a program can receive on that channel. It is an open question whether this loss of discrimination has material effect: so far, in a result due to Merro [40, 42], equators have only been shown able to encode the fusion calculus when we surrender that discrimination by disallowing output guards. The other significant work on equators, also by Merro [41], concerns their use with respect to program equivalence: they can be used to give a co-inductive characterisation of weak barbed congruence, again in the case where output guards are disallowed. In contrast, in the explicit fusion calculus, explicit fusions allow for an unrelated co-inductive characterisation of strong barbed congruence (Section 3.7); however, combining them with equators makes the problem of weak barbed congruence more difficult (Section 3.9).

Honda has proposed a general framework [30] to underly a range of calculi which use names. A basic part of this framework, partly inspired by his earlier work on equators, is a *wire*. Unlike equators, and like explicit fusions, the wire has immediate effect through structural congruence rather than reaction. Unlike both, the wire is such a fundamental part of the framework that even substitution is defined merely as the presence of a wire—rather than a consequence of it. Also, in order that the set of free names of a term should not be altered by structural congruence, the framework uses *degenerate wires*: these are like identity fusions $x{=}x$ but cannot be dismissed. Honda's work is intended as a general algebraic and graphical framework encompassing many calculi. Perhaps the explicit fusion calculus might fit conveniently into this framework, and the bisimulation theory of the following chapter might provide a case-study for the as-yet undeveloped theory of bisimulation in Honda's framework.

Explicit fusions have influenced recent work on *minimal reaction contexts*. For two programs to be equivalent we require that, if the first is able to execute in some context, the other one is also able. In fact, we require the same behaviour in *all* contexts. The task of checking all contexts is an arduous one. However, if we could find some class of minimal contexts, then perhaps we need not consider all contexts. Leifer, Luca, Milner and Sewell have attempted to systematically find the minimal contexts [37, 63]. As an example, consider the pi calculus term $\overline{x}.P \mid y.Q$. Current work by Milner [46] uses the fusion $x{=}y$ as a minimal context for this term. This is something on an anomaly—although the fusion is a context to allow reaction, it is not a valid context for pi calculus terms. In fact, Milner's work is set within a general categorical and graphical framework encompassing many calculi, and it is because of this generality that it finds itself not limited to pi contexts. (It was an earlier version of Milner's framework [44] that partly inspired Honda's framework, above). Milner's current work on minimal contexts is a first part of the project to develop the theory of bisimulation for his framework. The theory for the explicit fusion calculus, developed in the following chapter, provides a benchmark; we return to the issue technically in Section 3.5.

Chapter 3

Bisimulation for the explicit fusion calculus

In this chapter I develop the theory of bisimulation for the explicit fusion calculus. Bisimulation is a standard technique, but must be customised for each calculus for which it is used. Many definitions in this chapter are therefore variations of definitions that have appeared elsewhere, and some of the theorems are variations of theorems used for other calculi.

The chief accomplishments of this chapter are to show that barbed congruence and ground congruence coincide for the explicit fusion calculus, and to provide an efficient characterisation of them. Also to establish that, up to weak bisimulation, equators behave like explicit fusions. The novel technical contributions include a generalisation of Milner's concretions [45] and an 'ask' fusion transition.

The plan of the chapter is as follows. It is intended to be read in order: each section depends on almost all previous sections.

3.1 *Overview.* This section is a brief introduction to bisimulation.

3.2 *Labels and interfaces.* We give a labelled transition system for the explicit fusion calculus. It is simpler than that for the fusion calculus; nevertheless (Chapter 4) it yields the same equivalence. This section's version of the transition system is defined over structurally congruent classes of terms; an equivalent characterisation, but defined over terms themselves, is given in Section 3.6.

3.3 *Ground bisimulation.* We define strong ground bisimulation and congruence. They are similar in spirit to the very late bisimulation of Sangiorgi [59], also known as open bisimulation. We state that for two programs to be ground congruent, to have the same behaviour in all contexts, it is necessary and sufficient that their explicit fusions generate the same equivalence relation ('they have the same explicit fusions on the inside') and they have the same behaviour in all explicit fusion contexts ('explicit fusions on the outside'). Sections 3.4 to 3.7 introduce the techniques necessary to prove this statement, and culminate in the proof.

3.4 *Fusion transitions.* We introduce the 'ask' fusion label, which indicates that a term can react if provided with an explicit fusion. It is similar to a label used by Sangiorgi for his efficient characterisation of pi calculus open bisimulation. A variant of the label has appeared in recent work by Milner [46]. It is not the same as the 'tell' fusion labels used in the fusion and chi calculi.

3.5 *Efficient characterisation.* We give an efficient characterisation of ground congruence: that is, a definition that does not involve quantifying over all contexts. It uses the 'ask' fusion labels.

3.6 *Structural labels.* We give a compositional characterisation of the labelled transition system. With this we can deduce a term's labelled transitions through induction on its structure. Our compositional characterisation uses the 'ask' fusion labels.

3.7 *Ground congruence.* Proofs of congruence. We prove the theorem stated in Section 3.3, which characterises necessary and sufficient conditions for congruence. We also prove that the efficient characterisation in Section 3.5 is the same as ground congruence.

3.8 *Barbed bisimulation.* We define barbed bisimulation and barbed congruence. The congruence turns out to be the same as ground congruence.

3.9 *Equators.* We define weak bisimulation and equators. Up to weak bisimulation, equators are like explicit fusions: they allow names to be interchanged.

3.1 Overview of bisimulation

Bisimulation has become a standard technique for characterising the behaviour of programs in concurrent calculi. This section is a brief overview of bisimulation.

A software engineer needs to be able to tell whether two subroutines have the same behaviour: if they have, then the faster one can be used instead of the slower for instance. An optimising compiler must also be sure that none of its optimisations alter the behaviour of the program.

The problem is to define 'behaviour'. In a functional language it is easy: two programs have the same behaviour if and only if they give the same output for all possible inputs (ignoring for the moment the issue of non-termination). We might simply say that their behaviour *is* that mathematical object, the function from inputs to outputs. If one program calls the other, the overall behaviour is simply the composition of their two functions. In an interactive and concurrent language, however, it is much harder to define behaviour. This is because two programs in parallel may interact together in subtle ways: it is hard to characterise the composition of their behaviours.

A common approach is *bisimulation.* For two programs to be bisimilar, they must be able to perform the same external inputs and outputs at every stage in their execution. If one is able to send a free or locally created name, then the other must also be able to send the same free name or *any* local name. If one receives a free or local name, then the other must receive the same free

name or any local name. (This symmetry between sending and receiving is less apparent in the pi calculus, where free names are never received). A *bisimulation relation* is a relation that contains only bisimilar programs. The union of all bisimulations is itself a bisimulation; we call it *the* bisimulation. It is defined for the explicit fusion calculus in Section 3.3.

Our goal is to allow a faster implementation to be used in place of a slower one, so long as the two have the same behaviour. On the other hand, one might say that speed of execution is itself an aspect of behaviour. We say that two programs are *strongly bisimilar* if they have the same speed (measured as the number of internal reaction steps), or *weakly bisimilar* when we allow different speeds. The weak version has greater practical use, but is technically more awkward. We discuss weak bisimulation in Sections 3.9, show that equators behave like fusions in the weak case, and show how the awkwardness arises.

It is not in general possible to tell when two programs have the same behaviour. We must settle for two less ambitious goals: first to identify the platonic ideal of program equivalence, and second to come up with a sound approximation. For the engineer, it should be an approximation that is easy to use in proofs. For the compiler, it must be a decidable approximation. For our ideal, we take strong bisimulation. We discuss possible approximations below.

It is not enough just to know that two subprograms have the same behaviour themselves. We also need to know that in any larger program (context) in which the subprograms are used, that larger program will still have the same behaviour no matter which subprogram is used. Whenever an equivalence holds in all contexts in this way, we call it a *congruence*. The efficient bisimulation given in Section 3.5 is a sound and complete characterisation for strong bisimulation congruence. Efficient bisimulation avoids an infinite quantification over contexts and so is easier to use in proofs than the congruence. Efficient bisimulation is also decidable for programs with finitely many states.

For a concrete example, in a conventional imperative language, consider the following two implementations of a 'swap' program. These are written in ML. In them, the names x and y are pointers to integer variables, ! de-references a pointer, := assigns a new value, and **xor** is a bitwise exclusive-or operator.

```
fun swap x y = (                fun swap x y = (
    x := !x xor !y ;                let val z = !x in
    y := !x xor !y ;                x := !y ;
    x := !x xor !y );               y := z end );
```

The version on the left avoids using a temporary value. However, it fails in a context where x and y point to the same variable, such as the context $x{=}y$. The two programs are therefore not congruent.

Not all applications require subprograms to behave the same in *all* contexts. For instance, an optimising compiler often knows about the context in which the optimised fragment will be placed. And typically an engineer will expect a subprogram to be used only in a context which fulfills particular pre-conditions. The *distinction-full bisimulation* of Sangiorgi [59] (formerly known as indexed bisimulation) allows one to specify only contexts in which two names are different. Sometimes, for concurrent systems, we are only interested in contexts which execute concurrently with the subprogram: a relation is called a *behavioural equivalence* rather than a congruence when limited to these contexts.

Congruence alone is not sufficient in concurrent systems. After one program has executed for a time, a user might change the context by starting another program, or by physically connecting two networks together. If the concurrent system is also distributed, then a program might also be packaged up and shipped for further execution to a new location and a new context. We are therefore more interested in a version of congruence where, even after some execution steps, the resulting programs are still congruent. This is called *reduction-closed congruence*.

Reduction-closed congruence seems more appropriate for concurrent systems. However, many authors only define their relations to be context-closed before any execution has taken place—a *shallow* form of congruence.

A bisimulation might be too *large* to be useful, relating things that have the same behaviour even though they might differ in particular contexts. That is why congruence is important. On the other hand, a bisimulation might be too *small* to be useful—for instance, the identity relation is a bisimulation and a congruence, but it does not help us compare programs. Fortunately it turns out that the infinite union of all bisimulation congruences is itself a bisimulation congruence: two programs are interchangeable in all contexts if and only if they are related by this largest bisimulation congruence. It is neither too small nor too large.

Observational equivalence is another approach. By a twist of perspective, we ask whether an observer program can discriminate between two particular programs. The programs are said to be observationally equivalent if and only if no observer can distinguish them. Normally we assume that an observer can observe the sending and receiving of free and local names, and can observe only this. If the observer is allowed to grow, then observational equivalence is the same as reduction-closed behavioural equivalence. If the observer is kept fixed, then observational equivalence is the same as shallow behavioural equivalence.

One might also consider the stronger observational *congruence*, where the observer is able to prefix or replicate or restrict the program it observes. This corresponds to reduction-closed congruence or to shallow congruence.

The ability to send or receive a name is expressed as a four-tuple written $P \xrightarrow{\mu} I : P'$. This means that a program in state P can perform an output or input command over channel μ; it will send or receive the data I and end up in state P'. (We define these four-tuples formally in Section 3.2). For two programs P and Q to be bisimilar as described above, then for every transition that P can make, the program Q must also make the same transition, so that the results of both are bisimilar; and likewise P must match Q's transitions. To show that two programs are bisimilar, the typical proof technique is to exhibit a relation; then establish that, for any two related programs, they perform matching transitions such that the resulting programs are also in the relation. If this can be shown for all related programs, then the relation is a bisimulation.

We will write $P \xrightarrow{u} (x)(x : P')$ for a local name x being received over channel u. Equivalently, $P \xrightarrow{u} I : P'$ where $I = (x)(x : \_)$. Note that the scope of the name x includes P', and that it can be alpha-renamed.

There is ambiguity in the previous paragraph, according to when x is instantiated. The possibilities are, in increasing order of strength:

- *(Ground)* If P is bisimilar to Q and $P \xrightarrow{u} (x)(x : P')$ then there exists a

Q' such that $Q \xrightarrow{u} (x)(x : Q')$ and P' is bisimilar to Q'.

- *(Early)* If P is bisimilar to Q and $P \xrightarrow{u} (x)(x : P')$ then for all y there exists a Q' such that $Q \xrightarrow{u} (x)(x : Q')$ and $P'\{y/x\}$ is bisimilar to $Q'\{y/x\}$;
- *(Late)* If P is bisimilar to Q and $P \xrightarrow{u} (x)(x : P')$ then there exists a Q' such that $Q \xrightarrow{u} (x)(x : Q')$ and, for all y, $P'\{y/x\}$ is bisimilar to $Q'\{y/x\}$;
- *(Very late)* If P is bisimilar to Q and $P \xrightarrow{u} (x)(x : P')$ then there exists a Q' such that $Q \xrightarrow{u} (x)(x : Q')$ and P' is bisimilar to Q'; and moreover, whenever any P is bisimilar to any Q, then for all substitutions σ, $P\sigma$ is also bisimilar to $Q\sigma$.
- *(Ground congruence)* As for ground, and moreover whenever any P is bisimilar to any Q, then for all contexts E, $E[P]$ is bisimilar to $E[Q]$.

The difference between these possibilities are subtle, and to some extent depend on the calculus for which they are being used.

Ground, early and late bisimulations are too large to be congruences: they deem some programs equivalent that a context can distinguish. However, if we disallow contexts from prefixing a term, and if we are in the pi calculus, then early and late bisimulation are congruences.

Very late bisimulation is too large in the explicit fusion calculus, but is smaller than necessary in the pi calculus. That is to say, in the pi calculus, there are some programs which no context can distinguish but which are not related by very late bisimulation. Consider the example $(x)(\overline{u}x.P)$. In the pi calculus, no other name will ever substitute for x inside P, and so we should not quantify over substitutions that involve x. Sangiorgi uses his distinction-full bisimulation to respect this fact, thereby arriving at a larger congruence. Distinctions are not needed for the explicit fusion calculus, however, since for example the program $uy \mid (x)(\overline{u}x.P)$ can substitute y for x in P. Thus, explicit fusion contexts are more discriminating than pi contexts.

Ground congruence is by definition the largest ground bisimulation that is a congruence, but the quantification over contexts makes it an awkward definition to work with.

The choice of whether to use early, late or very late seems complicated and arbitrary. In an attempt to simplify matters for the pi calculus, Sangiorgi and Milner introduced *barbed bisimulation* [48]. This simply ignores the data that is transmitted, and ignores the resulting state for all but tau transitions. It turns out, fortunately, that the largest barbed bisimulation congruence is equal to the largest ground bisimulation congruence—both in the explicit fusion calculus and the pi calculus. We will write *barbs* as, for example, $\overline{u}x.P \xrightarrow{\overline{u}}$, sharing notation with labelled transitions; other authors write them as $\overline{u}x.P \downarrow \overline{u}$.

3.2 Labels and interfaces

Let the *labelled transition relation* be a set containing four-tuples $P \xrightarrow{\alpha} I : P'$. This four-tuple means that a program in state P can make the commitment α to react, communicating the data I and ending in state P'. For example, a program $\overline{u}x.P$ can commit to $\overline{u}$. After the commitment, what is left is the *concretion*

$x : P$. A program such as $\overline{u}.P \mid u.Q$ might also commit to an internal reaction $\overline{u}.P \mid u.Q \xrightarrow{\tau} \emptyset : P \mid Q$. We normally write this as just $P \mid Q$ since there is no data to communicate. The *labelled transitions* are the four-tuples, and the *label* is whatever expression is written above the arrow: for the moment, it is the same as the commitment. Although the entire set of four-tuples is commonly called a relation, the standard properties of relations such as reflexivity, symmetry and transitivity only apply to pairs (P, P') related by $P \xrightarrow{\tau} \emptyset : P'$. (Many authors use the notation $P \xrightarrow{\mu I} P'$ instead of $P \xrightarrow{\mu} I : P$).

Definition 9 (Commitments) *The set of commitments is* $\{\tau\} \cup \mathcal{N} \cup \overline{\mathcal{N}}$.

Let α range over commitments, and μ over non-tau commitments. We write $x \notin \alpha$ to mean that α is neither $\overline{x}$ nor x.

The concretions we will use here are a symmetric generalisation of those used by Milner for the pi calculus [45]. The pi calculus distinguishes between wholly-binding input concretions $(\tilde{x})(\tilde{x} : P)$ called *abstractions*, and possibly-binding output concretions $(\tilde{y})(\tilde{x} : P)$ with $\tilde{y} \subseteq \tilde{x}$. But the explicit fusion calculus has non-binding input, so both input and output commands lead to the same possibly-binding concretions. (Milner uses the notation $(\tilde{x})P$ for abstractions and $\nu\tilde{y}\langle\tilde{x}\rangle.P$ for concretions).

Definition 10 (Concretion) *A concretion has the form* $(\tilde{x})(\tilde{y} : P)$ *where the names in* $\tilde{x}$ *are distinct and contained in* $\tilde{y}$*, and no* $x \in \tilde{x}$ *is fused by* $\mathrm{Eq}(P)$.

Let C, D range over concretions. For a concretion $(\tilde{x})(\tilde{y} : P)$, we call the context $I = (\tilde{x})(\tilde{y} : \_)$ the *interface* of the concretion, and we write the concretion as $I : P$. The names $\tilde{x}$ are bound in this concretion.

Concretions will be used up to structural congruence. This is to express the precise conditions under which one transmitted datum is the same as another: a free name matches the same free name, and any bound name matches any other bound name. The rules for structural congruence are a little more complicated here than they are for pi calculus concretions. This is because of the possibility that explicit fusions might affect the names in a concretion.

The side condition to the above definition—that no $x \in \tilde{x}$ is fused by $\mathrm{Eq}(P)$—simplifies this problem. It will mean that when two concretions $I_1 : P_1$ and $I_2 : P_2$ are structurally congruent, then their contents are structurally congruent. (We will use structural congruence on concretions, as well as some operators, in presenting a labelled transition system for the explicit fusion calculus.)

The definition of structural congruence is awkward; we follow it with some illustrative examples.

Definition 11 (Structural congruence) *The structural congruence on concretions is defined by:* $(\tilde{x}_1)(\tilde{y}_1 : P_1) \equiv (\tilde{x}_2)(\tilde{y}_2 : P_2)$ *if and only if there exist fresh distinct* $\tilde{x}$ *of the same length, and permutations* π_1 *and* π_2*, and substitutions* $\sigma_1 = \{\tilde{x}/\pi_1\tilde{x}\}$ *and* $\sigma_2 = \{\tilde{x}/\pi_2\tilde{x}\}$ *such that* $P_1\sigma_1 \equiv P_2\sigma_2$ *and* $\tilde{y}_1\sigma_1$ *is identical to* $\tilde{y}_2\sigma_2$ *up to* $\mathrm{Eq}(P_1)$.

For example:

$$\begin{array}{rcll}
(x)(xy : P) & \equiv & (x)(xy : Q) \text{ if } P \equiv Q & \textit{Structural congruence on terms} \\
(x)(xy : P) & \equiv & (z)(zy : P\{z/x\}) & \textit{Alpha-rename bound names} \\
(xy)(xyz : P) & \equiv & (yx)(xyz : P) & \textit{Reorder bound names} \\
(x)(xy : P|y{=}z) & \equiv & (x)(xz : P|y{=}z) & \textit{Fusion-interchange free names}
\end{array}$$

There is no rule to fusion-interchange bound names. This is because terms would would allow it, such as $(x)(xy : P|x{=}z)$, we defined not to be valid concretions: no bound names may be fused. We would instead write this as $zy : (x)(P|x{=}z)$.

Because of the condition that no bound names are fused, and because $P_1\sigma_1 \equiv P_2\sigma_2$, then $\mathrm{Eq}(P_1) = \mathrm{Eq}(P_2)$. Therefore, the definition is symmetric.

Restriction and parallel composition on concretions are straightforward. We also use the *application* operator @. When one program commits to sending data and another commits to receiving data then the result is an application of their two concretions.

Definition 12 (Operators) *Restriction, composition and application of concretions are as follows. Assume by alpha-renaming that $\tilde{x}_1$ and $\tilde{x}_2$ do not intersect, and $\tilde{x}_1$ binds no name free in P_2 and $\tilde{x}_2$ binds no names free in P_1.*

$$(z)\ (\tilde{x})(\tilde{y} : P) \stackrel{def}{=} \begin{cases} (\tilde{x})(\tilde{y} : P) & \textit{if } z \in \{\tilde{x}\} \\ (z\tilde{x})(\tilde{y} : P) & \textit{if } z \in \{\tilde{y}\} - \{\tilde{x}\} \\ (\tilde{x})(\tilde{y} : (z)P) & \textit{otherwise} \end{cases}$$

$$(\tilde{x}_1)(\tilde{y}_1{:}P_1) \mid (\tilde{x}_2)(\tilde{y}_2{:}P_2) \stackrel{def}{=} (\tilde{x}_1\tilde{x}_2)(\tilde{y}_1\tilde{y}_2 : P_1 \mid P_2)$$

$$(\tilde{x}_1)(\tilde{y}_1{:}P_1) \mathbin{@} (\tilde{x}_2)(\tilde{y}_2{:}P_2) \stackrel{def}{=} (\tilde{x}_1\tilde{x}_2)(\tilde{y}_1{=}\tilde{y}_2 \mid P_1 \mid P_2)$$

We will at times have cause to treat terms as concretions, in order to write general rules that apply to both terms and concretions. In this case the term P stands for the concretion $()(\emptyset : P)$.

The following definition gives the labelled transitions a term might undergo. This is used in ground bisimulation: if one term undergoes a labelled transition, then the other term must undergo the same labelled transition. In the definition, recall that C and D range over concretions.

Definition 13 (Labelled transitions) *The labelled transition relation $P \xrightarrow{\alpha} I : P'$ is the smallest relation satisfying*

$$\mu\tilde{x}.P \xrightarrow{\mu} \tilde{x} : P$$

$$\overline{u}\tilde{x}.P \mid u\tilde{y}.Q \xrightarrow{\tau} \tilde{x}{=}\tilde{y} \mid P \mid Q$$

$$P \mid Q \xrightarrow{\alpha} C \mid Q \quad \textit{if } P \xrightarrow{\alpha} C$$

$$(x)P \xrightarrow{\alpha} (x)C \quad \textit{if } P \xrightarrow{\alpha} C \textit{ and } x \notin \alpha$$

$$Q \xrightarrow{\alpha} D \quad \textit{if } Q \equiv P \xrightarrow{\alpha} C \equiv D$$

Note that $\xrightarrow{\tau}$ is exactly the same as the reaction relation (Definition 4, page 24).

The following lemma shows that transitions never un-fuse any names that have been fused.

Lemma 14 *If* $P \xrightarrow{\alpha} I : P'$ *then* $\mathrm{Eq}(P) \subseteq \mathrm{Eq}(P')$.

Proof. By rule induction on the derivation of $P \xrightarrow{\alpha} I : P'$, using Corollary 8 (page 25) for the case of structural congruence. □

3.3 Ground bisimulation

We now define ground bisimulation $\dot{\sim}_g$ and reduction-closed ground congruence $\sim_g$.

The definition of ground congruence simply states the property that the relation is closed under all contexts. In practice, this is too hard a property to work with. It turns out that there is a simpler property which turns out to be equivalent: two terms are ground congruent if and only if they are ground bisimilar, and have the same explicit fusions on the inside, and behave the same with any explicit fusions on the outside. This property was inspired by the hero of *Le Petit Prince* [12], who made a study of elephants on the inside and outside of boa constrictors. The section concludes with a definition of this inside-outside bisimulation $\overset{\text{io}}{\dot{\sim}}_g$; the proof that it is equivalent to ground congruence is substantial and will take several more sections.

Definition 15 (Ground bisimulation) *A ground bisimulation is a relation* $\mathcal{S}$ *on terms such that if* $P \mathcal{S} Q$ *then, assuming* I *binds no names free in* P *or* Q,

- $P \xrightarrow{\alpha} I : P'$ *implies* $Q \xrightarrow{\alpha} I : Q'$ *and* $P' \mathcal{S} Q'$*; and*
- $Q \xrightarrow{\alpha} I : Q'$ *implies* $P \xrightarrow{\alpha} I : P'$ *and* $P' \mathcal{S} Q'$.

The infinite union $\dot{\sim}_g$ of all ground bisimulations, defined $\dot{\sim}_g = \{(P, Q) : \exists\, \mathcal{S}: P \mathcal{S} Q\}$ is itself a ground bisimulation, and therefore the largest ground bisimulation. We call it just *the* ground bisimulation. It is also an equivalence relation: it is transitive because, from any two ground bisimulations $\mathcal{S}_1$ and $\mathcal{S}_2$, their transitive closure $\mathcal{S} = \{(P, R) : \exists Q : P\ \mathcal{S}_1\ Q\ \mathcal{S}_2\ R\}$ is also a ground bisimulation. It is reflexive because every term has the same transitions as itself. And it is symmetric through symmetry of the definition.

Ground bisimulation is not a congruence—i.e. it is not closed with respect to contexts. This is why not:

(1) The programs $u{=}v$ and $\mathbf{0}$ are ground bisimilar, since neither undergoes any transitions. But consider them inside the context $\_ \mid \overline{u} \mid v$. The first allows a reaction; the second does not. *Two congruent terms must necessarily contain the same explicit fusions inside.*

(2) For our second counter-example we will use the summation operation, which allows a choice of which summand to use. We do this because summation allows for a succinct counter-example. For an analogous example which does not use summation, see Example 68. Now the programs $\overline{u} \mid v$ and $\overline{u}.v + v.\overline{u}$ are ground bisimilar. But consider them inside the context $\_ \mid u{=}v$. The first might undergo a reaction; the second will not. *Two congruent terms must necessarily behave the same with explicit fusions outside.* Indeed, this just is a special case of the definition of congruence.

Definition 16 (Ground congruence) *A ground bisimulation* $\mathcal{S}$ *is a reduction-closed ground congruence if whenever* $P \mathcal{S} Q$ *then*

- *for all contexts* E, $E[P]\ \mathcal{S}\ E[Q]$.

The largest reduction-closed ground congruence $\sim_g$ exists and is an equivalence.

This definition of congruence is an awkward one to use, since it involves quantifying over all contexts. Happily, the two necessary conditions for congruence given above also turn out to be sufficient. This allows for a more convenient characterisation:

Definition 17 (Inside-outside) *A ground bisimulation* $\mathcal{S}$ *is an inside-outside bisimulation iff whenever* $P\ \mathcal{S}\ Q$ *then*

- $\mathrm{Eq}(P) = \mathrm{Eq}(Q)$,
- *for all fusions* ϕ: $\phi|P\ \mathcal{S}\ \phi|Q$.

The largest inside-outside bisimulation $\overset{\text{io}}{\sim}_g$ exists, and is equal to ground congruence:

Theorem 18 $P \sim_g Q$ *if and only if* $P \overset{\text{io}}{\sim}_g Q$.

Proof. The forward direction is as follows. There are two properties from Definition 17 to prove. The second is an immediate consequence of the fact that $\mathcal{S}$ is a congruence. For the first, suppose the contrary: there exist u, v such that $(u, v) \in \mathrm{Eq}(P)$ but $(u, v) \notin \mathrm{Eq}(Q)$. Let x, y be names not occurring in P or Q. Consider the example $P \mid \overline{u}x \mid vy \mid \overline{x}$. This undergoes the transitions

$$P \mid \overline{u}x \mid vy \mid \overline{x} \xrightarrow{\tau} P \mid x{=}y \mid \overline{x} \xrightarrow{\overline{y}} P \mid x{=}y$$

But no single tau transition in Q can result in $x{=}y$; therefore no $\xrightarrow{\overline{y}}$ can follow a single tau transition; therefore $\mathcal{S}$ is not congruence. This contradicts the assumption that $\mathcal{S}$ is a $\sim_g$. Therefore, $\mathrm{Eq}(P) = \mathrm{Eq}(Q)$.

The reverse direction involves proving that $\overset{\text{io}}{\sim}_g$ is a congruence. It is substantially more complicated, involving an assortment of new techniques. These are introduced in Sections 3.4 to 3.6. The proof will eventually be completed in Section 3.7. □

3.4 Fusion transitions

This section describes a type of labelled transition called an *'ask' fusion transition*. It simplifies the task of testing for congruence: Theorem 18 has already said that we need only test over fusion contexts rather than all contexts; and with this label we need not even test over fusion contexts. A form of ask transitions was first used by Sangiorgi for the pi calculus. The ask fusion transitions should not be confused with the 'tell' fusion transitions of the fusion and chi calculi; we discuss the difference in Section 3.6.

Consider the ask transition

$$\overline{u}x.P \mid vy.Q \quad \xrightarrow{?u=v} \quad x{=}y \mid P \mid Q.$$

We have presented a notation without yet explaining its meaning. There are two meanings that might be understood: either it tells about a program's *syntax*, or it tells about the program's *behaviour*.

- *(Syntactical)* This program contains an output on the free channel u and an input on v, or vice versa.
- *(Behavioural)* In the presence of a context $u{=}v \mid \_$, the program can react.

We will adopt the syntactical meaning. The difference between syntax and behaviour is not always clear, since syntax always implies behaviour, and sometimes behaviour can only come from a particular syntax.

Let λ range over the labels $\{?x{=}y, \tau\} \cup \mathcal{N} \cup \overline{\mathcal{N}}$. We write $x \notin \lambda$ to mean that λ is neither $\overline{x}$, x, nor any fusion label involving x. We deem $?x{=}y$ equivalent to $?y{=}x$.

Definition 19 (Efficient labelled transitions) *The efficient labelled transition relation $P \xrightarrow{\lambda}_e I : P$ is the smallest relation satisfying*

$$\mu\tilde{x}.P \xrightarrow{\mu}_e \tilde{x} : P$$
$$\overline{u}\tilde{x}.P \mid u\tilde{y}.Q \xrightarrow{\tau}_e \tilde{x}{=}\tilde{y} \mid P \mid Q$$
$$\overline{u}\tilde{x}.P \mid v\tilde{y}.Q \xrightarrow{?u=v}_e \tilde{x}{=}\tilde{y} \mid P \mid Q$$

$$P \mid Q \xrightarrow{\lambda}_e C \mid Q \quad \text{if } P \xrightarrow{\lambda}_e C$$
$$(x)P \xrightarrow{\lambda}_e (x)C \quad \text{if } P \xrightarrow{\lambda}_e C \text{ and } x \notin \lambda$$
$$Q \xrightarrow{\lambda}_e D \quad \text{if } Q \equiv P \xrightarrow{\lambda}_e C \equiv D$$

In the third rule, u and v may be the same. For instance, $\overline{u}.P|u.Q \xrightarrow{?u=u}_e P|Q$.

The following lemma relates the transitions $\xrightarrow{\alpha}$ in the normal labelled transition system, to a subset of transitions $\xrightarrow{\lambda}_e$ in the efficient labelled transition system. Recall that α ranges over $\{\tau\} \cup \mathcal{N} \cup \overline{\mathcal{N}}$, while λ ranges over the same plus fusion labels.

Lemma 20 *$P \xrightarrow{\alpha} C$ if and only if $P \xrightarrow{\alpha}_e C$.*

Proof. $\longrightarrow_e$ has the same definition as $\longrightarrow$ (Definition 13, page 35) plus one extra rule introducing fusion transitions. Therefore the derivation of $P \xrightarrow{\alpha} P'$ has the same structure as the derivation of $P \xrightarrow{\alpha}_e P'$. □

3.5 Efficient bisimulation

Our overall larger goal is to define a bisimulation which generates the same relation as ground congruence—but without requiring an unwieldy quantification over all possible contexts. Inside-outside bisimulation (Definition 17, page 37) simplifies the task, by only requiring a quantification over fusion contexts $u{=}v \mid \_$. Now, using fusion labels, we will define an alternative characterisation of inside-outside bisimulation which avoids all context-quantifications entirely.

Consider the following property of inside-outside bisimulation:

- For all u and v, if $P \mathcal{S} Q$ and $u{=}v|P \xrightarrow{\alpha} I : P'$ then $u{=}v|Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{S} Q'$

We will use the efficient labelled transition system, and in particular the fusion transition $P \xrightarrow{?u=v}_e P'$, to express this without the need for the quantification. Recall however that our fusion transition actually declares more information about P: not just that it can react in a context $u{=}v \mid \_$ as required in the property, but also that it contains input and output commands. We will therefore remove this additional information:

- If $P \mathcal{S} Q$ and $P \xrightarrow{?u=v}_e P'$ then $u{=}v|Q \xrightarrow{\tau}_e Q'$ and $u{=}v|P' \mathcal{S} Q'$.

The extra information has been removed from the consequent, since $u{=}v|Q \xrightarrow{\tau}_e Q'$ says nothing about whether Q contains input and output commands on u or v.

Motivated by this connection with inside-outside bisimulation, we now define a new bisimulation $\overset{\text{e}}{\sim}_g$ which uses fusion labels. We call it the *efficient* bisimulation because it involves no infinite quantifications.

Definition 21 (Efficient bisimulation) *An efficient bisimulation is a symmetric relation $\mathcal{S}$ such that if $P \mathcal{S} Q$ then, assuming I binds no names free in Q,*

- $P \xrightarrow{\alpha}_e I : P'$ *implies* $Q \xrightarrow{\alpha}_e I : Q'$ *and* $P' \mathcal{S} Q'$,
- $P \xrightarrow{?u=v}_e P'$ *implies* $u{=}v|Q \xrightarrow{\tau}_e Q'$ *and* $u{=}v|P' \mathcal{S} Q'$,
- $\mathrm{Eq}(P) = \mathrm{Eq}(Q)$.

Our overall goal is to prove that inside-outside bisimulation is equal to ground congruence. Efficient bisimulation will provide an intermediary between the two:

Theorem 22 $P \overset{\text{io}}{\sim}_g Q$ *if and only if* $P \overset{\text{e}}{\sim}_g Q$.

The proof again is substantial, and will use a structural characterisation of $\xrightarrow{\mu}_e$ developed in the following section.

A slightly different ask fusion label has appeared recently in work by Milner [46], and it is interesting to compare it and its bisimulation to the work here.

Milner's fusion label $P \xrightarrow{u=v|\_} P'$ means that $u{=}v|\_$ is a minimal context needed to allow reaction: that is to say, $P \xrightarrow{?u=v} P'$ and $(u, v) \notin \mathrm{Eq}(P)$. It originates from a project [63] to use minimal contexts as a systematic way of generating labelled transitions and bisimulations. These systematically-generated bisimulations require that all labels match exactly:

- If $P \mathcal{S} Q$ and $P \xrightarrow{u=v|\_} P'$ then $Q \xrightarrow{u=v|\_} Q'$ and $u{=}v|P' \ \mathcal{S} \ u{=}v|Q'$.

This says: if P can *only* react in a context $u{=}v \mid \_$, then so too Q can only react in that context.

We might summarise the difference between our efficient bisimulation and Milner's bisimulation as follows.

- *(Efficient)* If one program can react in a context $u{=}v \mid \_$ then so can the other.

- *(Milner)* If one program can *only* react in that context, then so can the other.

Now we will see in Section 3.7 that efficient bisimulation is motivated by the fact that it is equal to ground congruence. However, it is not yet known whether Milner's bisimulation generates a familiar congruence. It is conceivable, though not yet proven, that the two bisimulations might generate the same congruence.

3.6 Structural labels

The labelled transition system given in Definition 19 (page 38) uses structural congruence. This makes it easy to understand the transitions, but awkward to enumerate all transitions possible from a given state. We now give an equivalent characterisation of the labelled transition system in which the labels of a term are deduced inductively on its structure.

The novel aspect is the use of the 'ask' fusion transitions introduced earlier. These are needed because of the following property of explicit fusions: in parallel with a program, they can allow tau transitions where perhaps none were possible before. For instance, $\overline{u}.P \mid v.Q$ has no tau transition in itself, but it does in the context $u{=}v \mid \_$. Therefore, the structured transition system needs to record: 'this part of the program has the potential for a tau transition, if it gets placed next to an explicit fusion'. Fusion transitions fulfil this need. (The fusion calculus also uses a fusion label to indicate a potential transition; but this is a different potential, and a different 'tell' fusion label, explained in Section 4.2. The pi calculus has no potential tau transitions—in it a term's transitions depend solely on the subcomponents of that term. Therefore the pi calculus does not need fusion transitions.)

Fusion transitions kill two birds with one stone. First, as discussed above, they allow for a structural labelled transition system by recording potentials. Second, as discussed in the previous section, they simplify the task of testing congruence by removing the need for an infinite quantification over fusion contexts. It is not clear to me whether these two birds are from the same bush.

We will write $P \vdash \lambda = \lambda'$ when P contains sufficient fusions to turn the label λ into λ'. The definition is given below. This notation generalises that of Definition 6, which only applied to names.

Definition 23 (Label equality) *The binary relation $P \vdash \_ = \_$ on labels is the least relation satisfying the following rules:*

$$
\begin{aligned}
P &\vdash ?u{=}v = ?v{=}u \\
P &\vdash x = y \quad \textit{if } (x,y) \in \mathrm{Eq}(P) \\
P &\vdash \overline{x} = \overline{y} \quad \textit{if } (x,y) \in \mathrm{Eq}(P) \\
P &\vdash ?x{=}y = ?u{=}v \quad \textit{if } (x,u) \in \mathrm{Eq}(P) \textit{ and } (y,v) \in \mathrm{Eq}(P).
\end{aligned}
$$

Definition 24 (Structured LTS) *The structured labelled transition system*

$P \xrightarrow{\lambda}_s I : P'$ is the smallest relation satisfying

$$\overline{u}\tilde{x}.P \xrightarrow{\overline{u}}_s \tilde{x} : P \qquad u\tilde{x}.P \xrightarrow{u}_s \tilde{x} : P$$

$$\frac{P \xrightarrow{\overline{u}}_s C \quad Q \xrightarrow{v}_s D}{P \mid Q \xrightarrow{?u=v}_s C@D} \qquad \frac{P \xrightarrow{u}_s C \quad Q \xrightarrow{\overline{v}}_s D}{P \mid Q \xrightarrow{?u=v}_s C@D} \qquad \frac{P \xrightarrow{?u=u}_s C}{P \xrightarrow{\tau}_s C}$$

$$\frac{P \xrightarrow{\lambda}_s C}{P \mid Q \xrightarrow{\lambda}_s C \mid Q} \qquad \frac{Q \xrightarrow{\lambda}_s D}{P \mid Q \xrightarrow{\lambda}_s P \mid D} \qquad \frac{P \xrightarrow{\lambda}_s C \quad P \vdash \lambda{=}\lambda'}{P \xrightarrow{\lambda'}_s C}$$

$$\frac{P \mid !P \xrightarrow{\lambda}_s C}{!P \xrightarrow{\lambda}_s C} \qquad \frac{P \xrightarrow{\lambda}_s C \quad x \notin \lambda}{(x)P \xrightarrow{\lambda}_s (x)C} \qquad \frac{P \xrightarrow{\lambda}_s C \equiv D}{P \xrightarrow{\lambda}_s D}$$

Recall that the application operator @ is defined in Section 3.2. The transition $P \xrightarrow{\overline{u}}_s C$ describes an output commitment that P can make, and the data it will send is recorded in the interface of the concretion C. The transition $Q \xrightarrow{\overline{u}}_s D$ describes an input commitment that Q can make. And the application $P|Q \xrightarrow{?u=v}_s C@D$ describes the consummation of those two commitments, and the concommitant exchange of data.

The following proposition fulfills our goal—of being able to enumerate all transitions possible from a given program.

Proposition 25 *If we have a transition $P \xrightarrow{\lambda}_s C$, then the transition is in fact one of the following:*

$\overline{u}\tilde{x}.Q \xrightarrow{\overline{u}}_s \equiv \tilde{x} : Q$

$u\tilde{x}.Q \xrightarrow{u}_s \equiv \tilde{x} : Q$

$(z)Q \xrightarrow{\lambda}_s \equiv (z)D$ *with* $Q \xrightarrow{\lambda}_s D,\ z \notin \lambda$

$(z)Q \xrightarrow{\tau}_s \equiv (z)Q'$ *with* $Q \xrightarrow{?u=u}_s Q'$

$!Q \xrightarrow{\lambda}_s \equiv D \mid !Q$ *with* $Q \xrightarrow{\lambda}_s D$

$!Q \xrightarrow{\tau}_s \equiv D_1@D_2 \mid !Q$ *with* $Q \xrightarrow{\overline{u}}_s D_1,\ Q \xrightarrow{u} D_2$

$!Q \xrightarrow{?u=v}_s \equiv D_1@D_2 \mid !Q$ *with* $Q \xrightarrow{\overline{u}}_s D_1,\ Q \xrightarrow{v} D_2$

$Q_1 \mid Q_2 \xrightarrow{\lambda}_s \equiv D_1 \mid Q_2$ *with* $Q_1 \xrightarrow{\lambda'}_s D_1,\ Q_1|Q_2 \vdash \lambda{=}\lambda'$

$Q_1 \mid Q_2 \xrightarrow{\lambda}_s \equiv Q_1 \mid D_2$ *with* $Q_2 \xrightarrow{\lambda'}_s D_2,\ Q_1|Q_2 \vdash \lambda{=}\lambda'$

$Q_1 \mid Q_2 \xrightarrow{\tau}_s \equiv Q_1' \mid Q_2$ *with* $Q_1 \xrightarrow{?u=v}_s Q_1',\ Q_1|Q_2 \vdash u{=}v$

$Q_1 \mid Q_2 \xrightarrow{\tau}_s \equiv Q_1 \mid Q_2'$ *with* $Q_2 \xrightarrow{?u=v}_s Q_2',\ Q_1|Q_2 \vdash u{=}v$

$Q_1 \mid Q_2 \xrightarrow{\tau}_s \equiv D_1@D_2$ *with* $Q_1 \xrightarrow{\overline{u}}_s D_1,\ Q_2 \xrightarrow{v}_s D_2,\ Q_1|Q_2 \vdash u{=}v$

$Q_1 \mid Q_2 \xrightarrow{\tau}_s \equiv D_1@D_2$ *with* $Q_1 \xrightarrow{u}_s D_1,\ Q_2 \xrightarrow{\overline{v}}_s D_2,\ Q_1|Q_2 \vdash u{=}v$

$Q_1 \mid Q_2 \xrightarrow{?u=v}_s \equiv D_1@D_2$ *with* $Q_1 \xrightarrow{\overline{x}}_s D_1,\ Q_2 \xrightarrow{y}_s D_2,\ Q_1|Q_2 \vdash ?u{=}v = ?x{=}y$

$Q_1 \mid Q_2 \xrightarrow{?u=v}_s \equiv D_1@D_2$ *with* $Q_1 \xrightarrow{x}_s D_1,\ Q_2 \xrightarrow{\overline{y}}_s D_2,\ Q_1|Q_2 \vdash ?u{=}v = ?x{=}y.$

Proof. For most terms and transitions, the proof involves a simple case analysis. For replication, the proof is by induction on the derivation of the transition. □

The following lemma and corollary establish that the structured transition system in this section is indeed just the same as the quotiented transition system in Section 3.2.

Lemma 26 *$P \equiv P_1 \xrightarrow{\lambda}_s C$ implies $P \xrightarrow{\lambda}_s C$*

Proof. For every rule in the structural congruence, we use Proposition 25 to analyse every possible transition taken by each side of the rule. We illustrate just with the structural congruence $(z)P \mid Q \equiv (z)(P|Q)$, for $z \notin \mathrm{fn}\, Q$. Suppose that $(z)(P|Q) \xrightarrow{\lambda}_s C$. Using Proposition 25 twice, for the restriction and then for the parallel composition, there are three cases for what the transition actually is. In all cases, we will prove that $(z)P \mid Q \xrightarrow{\lambda}_s C$ as well.

1. $(z)(P|Q) \xrightarrow{\lambda}_s \equiv (z)(D|Q)$ with $P \xrightarrow{\lambda'} D$, $P|Q \vdash \lambda = \lambda'$ and $z \notin \lambda$. Because z is not in $\mathrm{fn}\, Q$, and hence not in $\mathrm{Eq}(Q)$ either, there must exist some λ'' with $z \notin \lambda''$ and $P \vdash \lambda' = \lambda''$. For if there were not, then $\lambda' = z$, and neither P nor Q could convert it into any other equal label, and this would contradict the assumption that $P|Q \vdash \lambda = \lambda'$ with $z \notin \lambda$. It must also be the case that $(P\backslash z)|Q \vdash \lambda'' = \lambda$. Now we make a series of deductions:

$$
\begin{array}{rll}
P & \xrightarrow{\lambda'} & D \qquad \text{by assumption} \\
P & \xrightarrow{\lambda''} & D \qquad \text{since } P \vdash \lambda' = \lambda'' \\
(z)P & \xrightarrow{\lambda''} & (z)D \qquad \text{since } \lambda'' \text{ was assumed not equal to } z \\
(z)P \mid Q & \xrightarrow{\lambda''} & (z)D \mid Q \\
(z)P \mid Q & \xrightarrow{\lambda} & (z)D \mid Q \qquad \text{since } (P\backslash z)|Q \vdash \lambda'' = \lambda \\
(z)P \mid Q & \xrightarrow{\lambda} & (z)(D \mid Q) \qquad \text{since } (z)P \mid D \equiv (z)(P|D)
\end{array}
$$

2. $(z)(P|Q) \xrightarrow{\lambda}_s \equiv (z)(P|D)$ with $Q \xrightarrow{\lambda'} D$, $P|Q \vdash \lambda = \lambda'$ and $z \notin \lambda$. Now $z \notin \lambda'$ since $z \notin \mathrm{fn}\, Q$. Also since $P|Q \vdash \lambda' = \lambda$, and $z \notin \lambda, \lambda'$, then $(P|Q)\backslash z \vdash \lambda' = \lambda$. Moreover, $\mathrm{Eq}(Q) \subseteq \mathrm{Eq}(D)$, so $\mathrm{Eq}(P|D)\backslash z$ also makes $\lambda' = \lambda$. We now use these results:

$$
\begin{array}{rll}
Q & \xrightarrow{\lambda'} & D \qquad \text{by assumption} \\
(z)P \mid Q & \xrightarrow{\lambda'} & (z)P \mid D \\
(z)P \mid Q & \xrightarrow{\lambda'} & (z)(P \mid D) \qquad \text{since } (z)P \mid D \equiv (z)(P|D) \\
(z)P \mid Q & \xrightarrow{\lambda} & (z)(P \mid D) \qquad \text{since } (P|D)\backslash z \vdash \lambda' = \lambda
\end{array}
$$

3. $(z)(P|Q) \xrightarrow{?u=v} (z)(D_1 @ D_2)$ with $P \xrightarrow{\overline{x}} D_1$, $Q \xrightarrow{y} D_2$, $P|Q \vdash ?u=v = ?x=y$ and $z \notin \{u, v\}$. Therefore there exists a w such that $P \vdash x = w$ and

$w \neq z$. Moreover, $(P|Q)\backslash z \vdash ?u{=}v = ?w{=}y$.

$$
\begin{array}{rll}
P & \xrightarrow{\overline{w}} & D_1 \qquad \text{by assumption} \\
(z)P & \xrightarrow{\overline{w}} & (z)D_1 \\
(z)P \mid Q & \xrightarrow{?w=y} & (z)D_1 @ D_2 \\
(z)P \mid Q & \xrightarrow{?w=y} & (z)(D_1 @ D_2) \qquad \text{since } (z)D_1 @ D_2 = (z)(D_1 @ D_2) \\
(z)P \mid Q & \xrightarrow{?u=v} & (z)(D_1 @ D_2) \qquad \text{since } (P|Q)\backslash z \vdash ?u{=}v = ?w{=}y
\end{array}
$$

The other cases are degenerate or mirrored versions of these three. □

Corollary 27 *$P \xrightarrow{\lambda}_e P'$ if and only if $P \xrightarrow{\lambda}_s P'$.*

3.7 Ground congruence results

We are finally in a position to provide proofs for the two outstanding propositions: first that inside-outside bisimulation $\overset{\text{io}}{\sim}_g$ is equal to efficient bisimulation $\overset{\text{e}}{\sim}_g$; second that $\overset{\text{e}}{\sim}_g$ is a congruence. These will complete the proof for Proposition 18, that the two bisimulations are equal to ground congruence.

First, we need some technical results to relate fusion labels to explicit fusion transitions:

Lemma 28

1. *$P \xrightarrow{?u=v}_e P'$ implies $u{=}v|P \xrightarrow{\tau}_e u{=}v|P'$*
2. *If $Q|x{=}y \vdash w = z$ and $Q \vdash ?x{=}y = ?u{=}v$ then either $Q \vdash ?w{=}z = ?u{=}v$ or $Q \vdash w = z$.*
3. *If $P \vdash ?x{=}y = ?u{=}v$ then $P|x{=}y \equiv P|u{=}v$.*

Proof. The first part is by induction on the structure of P, using Proposition 25 for each case. The other two parts are straightforward manipulations of equivalence classes, using Corollary 8 and Definition 23. □

Proposition 29 *$P \overset{\text{io}}{\sim}_g Q$ if and only if $P \overset{\text{e}}{\sim}_g Q$.*

Proof. Compare the definition of inside-outside bisimulation $\overset{\text{io}}{\sim}_g$ (Definition 17, page 37) with that of efficient bisimulation $\overset{\text{e}}{\sim}_g$ (Definition 21, page 39). They are already very similar, so not much work is needed.

In the forward direction, given $\mathcal{S}{=}\overset{\text{io}}{\sim}_g$, we show that it is also an efficient bisimulation. From Lemma 20, the first parts of the definitions are equivalent. It remains to prove the second parts. Assume $P \xrightarrow{?u=v}_e P'$. From Lemma 28, $u{=}v \mid P \xrightarrow{\tau}_e u{=}v \mid P'$. Since $\mathcal{S}$ is a $\overset{\text{io}}{\sim}_g$, and from the first two parts of its definition, there exists a Q' such that $u{=}v\,|\,Q \xrightarrow{\tau}_e Q'$ and $u{=}v\,|\,P'\ \mathcal{S}\ Q'$.

In the reverse direction, we construct $\mathcal{S}{=}\{(\phi\,|\,P, \phi\,|\,Q).\ P \overset{\text{e}}{\sim}_g Q\}$ and prove that $\mathcal{S}$ is an inside-outside bisimulation. The second and third parts of the definition of $\overset{\text{io}}{\sim}_g$ are straightforwardly satisfied. It remains to prove the first.

Suppose that $\phi \mid P \xrightarrow{\mu}_e C$. From Proposition 25, the transition is actually $\phi \mid P \xrightarrow{\mu}_e \phi \mid I : P'$ with $P \xrightarrow{\mu'}_e I : P'$ and $\phi \mid P \vdash \mu = \mu'$. Since $P \overset{\mathrm{e}}{\sim}_g Q$ we deduce that $Q \xrightarrow{\mu'}_e I : Q'$ with $P' \overset{\mathrm{e}}{\sim}_g Q'$. From this the appropriate $\phi \mid Q$ transition can be constructed. □

We will now prove that $\overset{\mathrm{e}}{\sim}_g$ is a congruence.

Proposition 30 (Congruence) *If $P \overset{\mathrm{e}}{\sim}_g Q$, then there exists an efficient bisimulation $\mathcal{S}$ relating P and Q and which is a reduction-closed congruence.*

Proof. Construct a relation $\mathcal{S}$ which contains $\overset{\mathrm{e}}{\sim}_g$ and which is closed under the following conditions:

1. if $P \equiv P_1 \; \mathcal{S} \; Q_1 \equiv Q$ then $P \; \mathcal{S} \; Q$;
2. if $P \; \mathcal{S} \; Q$ then $(x)P \; \mathcal{S} \; (x)Q$, $\mu\tilde{x}.P \; \mathcal{S} \; \mu\tilde{x}.Q$ and $!P \; \mathcal{S} \; !Q$;
3. if $P_1 \; \mathcal{S} \; Q_1$ and $P_2 \; \mathcal{S} \; Q_2$ then $P_1|P_2 \; \mathcal{S} \; Q_1|Q_2$.

Clearly $\mathcal{S}$ is a reduction-closed congruence. It remains to prove that it is an efficient bisimulation, which we do by induction on the closure properties. But first, a comment on why the third closure condition is as strong as it is. Imagine the weaker condition that if $P \; \mathcal{S} \; Q$ then $P \mid R \; \mathcal{S} \; Q \mid R$ and $R \mid P \; \mathcal{S} \; R \mid Q$. Now consider the replication case, that $P \xrightarrow{\alpha} I : P'$ giving $!P \xrightarrow{\alpha} P' \mid !P$. We can deduce that $!Q \xrightarrow{\alpha} Q' \mid !Q$ and $P' \; \mathcal{S} \; Q'$. But now we need closure conditions on $\mathcal{S}$ that are strong enough to deduce that $P'|!P \; \mathcal{S} \; Q'|!Q$. The weaker conditions are not adequate. That is why we have used stronger conditions.

Now, we perform the induction on the closure properties of $\mathcal{S}$. The induction property is that two terms related by $\mathcal{S}$ fulfill the requirements of an efficient bisimulation.

1. Suppose $P \equiv P_1 \; \mathcal{S} \; Q_1 \equiv Q$, and suppose $P \xrightarrow{\alpha} I : P'$. Then (Definition 13) $P_1 \xrightarrow{\alpha} I : P'$. From the induction hypothesis and Definition 13, $Q \xrightarrow{\alpha} I : Q'$ and $P' \; \mathcal{S} \; Q'$, as desired. The case for fusion labels is similar. Finally, the induction hypothesis says that $\mathrm{Eq}(P_1) = \mathrm{Eq}(Q_1)$, and so (Lemma 6) we get $\mathrm{Eq}(P) = \mathrm{Eq}(Q)$.

2a. Suppose $(z)P \xrightarrow{\alpha} I : P'$. From Proposition 25, $P \xrightarrow{\alpha} J : P_1$ such that $z \notin \alpha$ and, using restriction on concretions (Definition 12), $(z)(J : P_1) \equiv I : P'$. From the induction hypothesis, $Q \xrightarrow{\alpha} J : Q_1$, and $P_1 \; \mathcal{S} \; Q_1$. Since $z \notin \alpha$ we get $(z)Q \xrightarrow{\alpha} (z)(J : Q_1)$. Since this restricted concretion is structurally congruent to $I : P'$ we get $(z)Q \xrightarrow{\alpha} I : Q'$ as desired, with $Q' \; \mathcal{S} \; P'$. The case for fusion transitions is similar. Finally, from the induction hypothesis and from the definition of $\mathrm{Eq}(\cdot)$, we get that $\mathrm{Eq}((x)P) = \mathrm{Eq}((x)Q)$ as desired.

2b. Suppose $\mu\tilde{x}.P \xrightarrow{\mu} \tilde{x} : P$. It is clear that $\mu\tilde{x}.Q \xrightarrow{\mu} \tilde{x} : Q$. Moreover, from the induction hypothesis, $P \; \mathcal{S} \; Q$ as desired. Finally, from the definition of $\mathrm{Eq}(\cdot)$, we get $\mathrm{Eq}(\mu\tilde{x}.P) = \mathrm{Eq}(\mu\tilde{x}.Q) = \mathbf{I}$.

2c. Suppose $!P \xrightarrow{\lambda} I : P_1$. From Proposition 25, there are three possibilities A, B and C for what this transition might be:

A1. $!P \xrightarrow{\alpha} I : (P' \mid !P)$ with $P \xrightarrow{\alpha} I : P'$. From the induction hypothesis, $Q \xrightarrow{\alpha} I : Q'$ and $P' \mathcal{S} Q'$. Since we assumed I not to clash, we get $!Q \xrightarrow{\alpha} I : (Q' \mid !Q)$ as desired. Finally we must show that the result is related by $\mathcal{S}$. Now $P \mathcal{S} Q$, and hence $!P \mathcal{S} !Q$ by the replication-closure of $\mathcal{S}$. And because $P' \mathcal{S} Q'$ we get $P'|!P \mathcal{S} Q'|!Q$ by the parallel-closure of $\mathcal{S}$.

A2. $!P \xrightarrow{?u=v} P' \mid !P$, with $P \xrightarrow{?u=v} P'$. From the induction hypothesis, $u{=}v|Q \xrightarrow{\tau} u{=}v|Q_2$ such that $u{=}v|P' \mathcal{S} u{=}v|Q_2$. By Proposition 25, the transition of $u{=}v \mid Q$ might have arisen in two ways. The first way is that $Q \xrightarrow{\tau} Q_2$; in this case, $!Q \xrightarrow{\tau} Q_2|!Q$ so that

$$u{=}v \mid !Q \xrightarrow{\tau} u{=}v \mid Q_2 \mid !Q. \tag{2}$$

Now we have $!P \mathcal{S} !Q$, and $u{=}v|P' \mathcal{S} u{=}v|Q_2$. Putting these together, using the closure properties of $\mathcal{S}$, we obtain the desired result:

$$u{=}v \mid P' \mid !P \quad \mathcal{S} \quad u{=}v \mid Q_2 \mid !Q. \tag{3}$$

The second way in which the transition of Q might have arisen is through $Q \xrightarrow{?x=y} Q_2$ with $u{=}v|Q \vdash x = y$. In this case, $!Q \xrightarrow{?x=y} Q_2|!Q$, which again yields Equation 2. And Equation 3 holds in the same way.

B. $!P \xrightarrow{?u=v} C@D \mid !P$ with $P \xrightarrow{\overline{u}} C$ and $P \xrightarrow{v} D$. The notation is a little cumbersome, so we introduce some abbreviations. Let $C = I : P' = (\tilde{x})(\tilde{y} : P')$ and $D = J : P'' = (\tilde{w})(\tilde{z} : P'')$. Then we write $I@J[P'|P''] = C@D = (\tilde{x}\tilde{w})(\tilde{y}{=}\tilde{z} \mid P' \mid P'')$. Note that $\tilde{x}$ and $\tilde{w}$ were assumed not to bind P'' or P' respectively. Therefore neither bind $!P$ and we can write $C@D|!P \equiv I@J[P'|P''|!P]$.

Thus, we rewrite this (B) possibility as $!P \xrightarrow{?u=v} I@J[P' \mid P'' \mid !P]$ with $P \xrightarrow{\overline{u}} I : P'$ and $P \xrightarrow{v} J : P''$. From the induction hypothesis, $Q \xrightarrow{\overline{u}} I : Q'$ and $Q \xrightarrow{v} J : Q''$ with $P' \mathcal{S} Q'$ and $P'' \mathcal{S} Q''$. Therefore,

$$!Q \xrightarrow{?u=v} I@J[Q' \mid Q'' \mid !Q].$$

By Lemma 28,

$$u{=}v \mid !Q \xrightarrow{\tau} u{=}v \mid I@J[Q' \mid Q'' \mid !Q]$$

as desired. We can use the closure properties of $\mathcal{S}$ to prove that

$$u{=}v \mid I@J[P' \mid P'' \mid !P] \quad \mathcal{S} \quad u{=}v \mid I@J[Q' \mid Q'' \mid !Q]$$

as desired. Note that the context $u{=}v \mid I@J[\_]$ amounts to some restrictions and parallel compositions, and so is dealt with by the restriction and parallel closure properties of $\mathcal{S}$.

C. $!P \xrightarrow{\tau} I@J[P'|P''|!P]$ with $P \xrightarrow{\overline{u}} I : P'$ and $P \xrightarrow{u} J : P''$. Then from the induction hypothesis, $Q \xrightarrow{\overline{u}} I : Q'$ and $Q \xrightarrow{u} J : Q''$ with $P' \mathcal{S} Q'$ and $P'' \mathcal{S} Q''$. From the induction hypothesis, Q also makes these transitions, and this case proceeds in much the same way as B2.

Finally, to conclude the replication case, $\mathrm{Eq}(!P) = \mathrm{Eq}(P) = \mathrm{Eq}(Q) = \mathrm{Eq}(!Q)$ by the induction hypothesis and the definition of $\mathrm{Eq}(\cdot)$.

3. Suppose $P_1 \mid P_2 \xrightarrow{\lambda} C$. For the equivalence relation, $\mathrm{Eq}(P_1|P_2) = \mathrm{Eq}(Q_1|Q_2)$ follows directly from the definition of $\mathrm{Eq}(\cdot)$ and the induction hypotheses. For the transitions, Proposition 25 lists eight possibilities. Discounting the symmetric possibilities leaves us with four cases, A, B, C and D.

A1. $P_1 \mid P_2 \xrightarrow{\alpha} C@P_2$ with $P_1 \xrightarrow{\alpha'} C$ and $P_1|P_2 \vdash \alpha' = \alpha$. Let $C = I : P_1'$ were I does not clash with P_2, Q_1 or Q_2. Then, through parallel composition on concretions, and since I does not clash, this A1 possibility is actually

$$P_1 \mid P_2 \xrightarrow{\alpha} I : (P_1' \mid P_2).$$

Q_1 undergoes the same transition $Q_1 \xrightarrow{\alpha'} I : Q_1'$; and $\mathrm{Eq}(Q_1|Q_2) = \mathrm{Eq}(P_1|P_2)$, so $Q_1|Q_2$ undergoes the transition

$$Q_1 \mid Q_2 \xrightarrow{\alpha} I : (Q_1' \mid Q_2)$$

as desired. The closure properties of $\mathcal{S}$ ensure that the results are related in $\mathcal{S}$, just as in the replication case. (The symmetric case, where P_2 does the transition, is similar).

A2. $P_1 \mid P_2 \xrightarrow{?u=v} P_1' \mid P_2$ with $P_1 \xrightarrow{?x=y} P_1'$ and $P_1|P_2 \vdash ?u=v = ?x=y$. By the induction hypothesis,

$$x=y \mid Q_1 \xrightarrow{\tau} x=y \mid Q_1' \text{ and } x=y \mid P_1' \mathcal{S} x=y \mid Q_1'.$$

Therefore $x=y \mid Q_1 \mid Q_2 \xrightarrow{\tau} x=y \mid Q_1' \mid Q_2$. But since $\mathrm{Eq}(Q_1|Q_2) = \mathrm{Eq}(P_1|P_2)$, and since $P_1|P_2 \vdash ?u=v = ?x=y$, then by Lemma 28 we get $u=v \mid Q_1 \mid Q_2 \xrightarrow{\tau} u=v \mid Q_1' \mid Q_2$ as desired. To show that

$$u=v \mid P_1' \mid P_2 \quad \mathcal{S} \quad u=v \mid Q_1' \mid Q_2$$

we use the fact that explicit fusions are monotonically increasing (Lemma 14): hence $\mathrm{Eq}(Q_1' \mid Q_2) \supseteq \mathrm{Eq}(Q_1 \mid Q_2)$. And since $Q_1|Q_2 \vdash u=v$, we get that $u=v \mid Q_1' \mid Q_2 \equiv Q_1' \mid Q_2$. The rest follows from the closure properties of $\mathcal{S}$. (The symmetric case, where P_2 does the transition, is similar.)

B. $P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2$ with $P_1 \xrightarrow{?x=y} P_1'$ and $P_1 \mid P_2 \vdash x = y$. From the induction hypothesis, $x=y \mid Q_1 \xrightarrow{\tau} x=y \mid Q_1'$ and $x=y \mid P_1' \mathcal{S} x=y \mid Q_1'$. Therefore, $x=y \mid Q_1 \mid Q_2 \xrightarrow{\tau} x=y \mid Q_1' \mid Q_2$ as desired. And the results are related by $\mathcal{S}$ just as in the previous case. (The symmetric case, where P_2 does the transition, is similar.)

C. $P_1 \mid P_2 \xrightarrow{\tau} I@J[P_1' \mid P_2']$ with $P_1 \xrightarrow{\overline{x}} I : P_1'$, and $P_2 \xrightarrow{y} J : P_2'$, and $P_1 \mid P_2 \vdash x = y$. Recall the notation $I@J[\cdot]$ from part 2c.B of this proof. By the induction hypothesis, $Q_1 \xrightarrow{\overline{x}} I : Q_1'$ and $Q_2 \xrightarrow{y} J : Q_2'$ with $P_1' \mathcal{S} Q_1'$ and $P_2' \mathcal{S} Q_2'$. Therefore $Q_1 \mid Q_2 \xrightarrow{?x=y} I@J[Q_1' \mid Q_2']$, so $x{=}y \mid Q_1 \mid Q_2 \xrightarrow{\tau} x{=}y \mid I@J[Q_1' \mid Q_2']$. Again, we can remove the $x{=}y$ through monotonicity of explicit fusions as in the previous cases, giving

$$I@J[P_1' \mid P_2'] \mathcal{S} I@J[Q_1' \mid Q_2']$$

as desired. (The symmetric case, where P_1 receives a name and P_2 sends a name, is similar.)

D. $P_1 \mid P_2 \xrightarrow{?u=v} I@J[P_1' \mid P_2']$. This and its symmetric form is are a combination of the previous three cases. □

At last we can complete the proof for Theorem 18 (page 37), that $P \sim_g Q$ if and only if $P \overset{\text{io}}{\sim}_g Q$. The forward case has already been proved; we now prove the reverse case.

Proof. From Proposition 30, $\overset{\text{e}}{\sim}_g$ is a reduction-closed congruence. From Proposition 29, $\overset{\text{io}}{\sim}_g$ is also a reduction-closed congruence. It therefore satisfies Definition 16 (page 36) and is part of $\sim_g$. □

Corollary 31 *$P \overset{\text{e}}{\sim}_g Q$ if and only if $P \sim_g Q$.*

This concludes our study of strong ground congruence. We have established an efficient characterisation of it, avoiding the need to quantify over infinitely many contexts. We have also provided a structural characterisation of labelled transitions. Together, these two results make strong ground congruence easier to use in proofs.

3.8 Barbed bisimulation

We now define barbed bisimulation $\dot{\sim}_b$ for the explicit fusion calculus, and reduction-closed barbed congruence $\sim_b$. In fact, we shall see that barbed congruence and ground congruence are equal. This gives us reason to believe that we have discovered the 'true' reduction-closed congruence for the explicit fusion calculus. Our belief will be further strengthened by the result in Section 4.3 (page 64), that barbed and ground congruence are also equal to the bisimulation independently proposed for the fusion calculus by Victor and Parrow [52].

We also define shallow barbed congruence $\sim_b^s$, which is only closed with respect to context at the start. The difference between reduction-closed congruence and shallow congruence was discussed in Section 3.1.

Barbed bisimulation is defined using an *observation relation*. This is a cut-down version of labelled transitions that discards some of the information. It is also defined using the reaction relation (Definition 4) whose definition we repeat here for convenience.

Definition 32 (Observation and reaction) *The observation relation* $P \xrightarrow{\mu}$ *is the smallest relation satisfying*

$$\mu\tilde{x}.P \xrightarrow{\mu}$$

$$\begin{aligned} P \mid Q &\xrightarrow{\mu} \quad \text{if } P \xrightarrow{\mu} \\ (x)P &\xrightarrow{\mu} \quad \text{if } P \xrightarrow{\mu} \text{ and } x \notin \mu \\ Q &\xrightarrow{\mu} \quad \text{if } Q \equiv P \xrightarrow{\mu} \end{aligned}$$

The internal reaction relation $P \xrightarrow{\tau} P'$ *is the smallest relation satisfying*

$$\overline{u}\tilde{x}.P \mid u\tilde{y}.Q \xrightarrow{\tau} \tilde{x}{=}\tilde{y} \mid P \mid Q$$

$$\begin{aligned} P \mid Q &\xrightarrow{\tau} P' \mid Q \quad \text{if } P \xrightarrow{\tau} P' \\ (x)P &\xrightarrow{\tau} (x)P' \quad \text{if } P \xrightarrow{\tau} P' \\ Q &\xrightarrow{\tau} Q' \quad \text{if } Q \equiv P \xrightarrow{\tau} P' \equiv Q' \end{aligned}$$

Definition 33 (Barbed bisimulation) *A barbed bisimulation is a relation* $\mathcal{S}$ *such that* $P \mathcal{S} Q$ *implies*

- *if* $P \xrightarrow{\mu}$ *then* $Q \xrightarrow{\mu}$*;*
- *if* $Q \xrightarrow{\mu}$ *then* $P \xrightarrow{\mu}$*;*
- *if* $P \xrightarrow{\tau} P'$ *then there exists a* Q' *such that* $Q \xrightarrow{\tau} Q'$ *and* $P' \mathcal{S} Q'$*;*
- *if* $Q \xrightarrow{\tau} Q'$ *then there exists a* P' *such that* $P \xrightarrow{\tau} P'$ *and* $P' \mathcal{S} Q'$*.*

The largest barbed bisimulation $\dot{\sim}_b$ exists and is an equivalence.

Definition 34 (Barbed congruence) *A barbed bisimulation* $\mathcal{S}$ *is a reduction-closed barbed congruence iff whenever* $P \mathcal{S} Q$ *then*

- *for all contexts* E*,* $E[P] \mathcal{S} E[Q]$*.*

The largest barbed congruence $\sim_b$ exists and is an equivalence.

Definition 35 (Shallow) *Shallow barbed congruence* $\sim_b^s$ *relates any two programs* P *and* Q *such that for all contexts* E*,* $E[P] \dot{\sim}_b E[Q]$*.*

Clearly, if two programs are related by reduction-closed barbed congruence, then they are related by shallow barbed congruence. I do not know whether the converse is true.

We will now prove that reduction-closed barbed congruence and ground congruence coincide. But first, a useful lemma:

Lemma 36 *If* P *and* Q *are related by a* $\sim_b$*, then* $\mathrm{Eq}(P) = \mathrm{Eq}(Q)$

Proof. As for Theorem 18. □

Proposition 37 *If* $P \sim_g Q$ *then* $P \sim_b Q$*.*

Proof. The proof is straightforward, since ground congruence $\sim_g$ (Definition 16, page 36) is stronger than that of barbed congruence $\sim_b$.

For the first part of Definition 33, suppose $P \xrightarrow{\mu}$. Then there exists an I and P' such that $P \xrightarrow{\mu} I : P'$. Therefore $Q \xrightarrow{\mu} I : Q'$, and hence $Q \xrightarrow{\mu}$. The tau case and congruence closure are trivial. □

Proposition 38 *If $P \sim_b Q$ then $P \sim_g Q$.*

Proof. The issue is that ground bisimulation $\sim_g$ pays attention to the data and resulting state of input and output actions, but barbed bisimulation $\sim_b$ discards that information. Our task is to reconstruct it. In particular, we must reconstruct the concretion $P \xrightarrow{\overline{u}} I : P'$ from the barb $P \xrightarrow{\overline{u}}$. We do this with a small context $u\tilde{y}.\phi$, where ϕ is a fusion of two fresh names. This will be our litmus paper: if one term does a reaction that uses the context, and hence liberates ϕ, then the other term's matching reaction must also liberate ϕ, and hence use the context. The bulk of the work is then to show that interfaces match. This amounts to picking apart the fusions involving $\tilde{y}$, which is possible since each $y \in \tilde{y}$ is fresh and distinct.

The largest barbed congruence relates P to Q. We will show that it is a ground bisimulation. Tau transitions are the same for ground as they are for barbed, so we need only consider input and output actions. In fact, we will only consider output actions, since input actions are the same.

Suppose that $P \xrightarrow{\mu}$; therefore $P \xrightarrow{\mu} (\tilde{x})(\tilde{u} : P')$ for some $\tilde{x}, \tilde{u}, P'$. Let us pick $\tilde{y}, w_1, w_2$ to be fresh and distinct. The fusion $w_1{=}w_2$ will be our litmus paper, as discussed. We can deduce the reaction

$$P \mid \overline{\mu}\tilde{y}.(w_1{=}w_2) \quad \xrightarrow{\tau} \quad (\tilde{x})(\tilde{y}{=}\tilde{u} \mid w_1{=}w_2 \mid P')$$

where $\tilde{y}$ does not occur in P'. Let us write P'' for the right hand side of this reaction. Since P and Q are related by a barbed congruence, we can deduce that Q makes a matching transition

$$Q \mid \overline{\mu}\tilde{y}.(w_1{=}w_2) \quad \xrightarrow{\tau} \quad Q''$$

with $P'' \; \mathcal{S} \; Q''$. In fact, we can say more about the structure of Q''. First, note that P'' contains a free fusion $w_1{=}w_2$; by Lemma 36, so does Q''. The only way this is possible is if the term $\overline{\mu}\tilde{y}.(w_1{=}w_2)$ participated in the reaction. In particular, there must be some $\tilde{z}, \tilde{v}$ such that

$$Q \xrightarrow{\overline{u}} (\tilde{z})(\tilde{v} : Q')$$

giving the reaction

$$Q \mid u\tilde{y}.(w_1{=}w_2) \quad \xrightarrow{\tau} \quad (\tilde{z})(\tilde{y}{=}\tilde{v} \mid w_1{=}w_2 \mid Q') \quad \equiv \quad Q''$$

where $\tilde{y}$ does not occur in Q'.

The next stage of the proof is to show that the interface $(\tilde{x})(\tilde{u} : \_)$ for P is equal to the interface $(\tilde{z})(\tilde{v} : \_)$ for Q. In outline: will show that, up to alpha-renaming and $\mathrm{Eq}(P')$ and $\mathrm{Eq}(Q')$, $\tilde{x} = \tilde{z}$ and $\tilde{u} = \tilde{v}$. We will use the fact that each name in $\tilde{y}$ is fresh and distinct. In essence, for each y_i, there are two possibilities: it might be fused to a free name u_i, in which case (Lemma 36)

the fusion $y_i{=}v_i$ in Q'' is the same; or it might be fused to a bound name u_i, in which case it must also be fused to a bound name in Q''.

To turn the above outline into a precise proof, we now pay more attention to equivalence classes of names. By Lemma 36, using the notation for equivalence relations (Definition 5, page 25),

$$(\tilde{y}{=}\tilde{u} \oplus \mathrm{Eq}(P'))\backslash\tilde{x} \quad = \quad (\tilde{y}{=}\tilde{v} \oplus \mathrm{Eq}(Q'))\backslash\tilde{z}.$$

To obtain this we have omitted the fusion $w_1{=}w_2$, since it is the same on both sides and the names w_1 and w_2 were assumed fresh. Recall also (Definition 10, page 34) that the names $\tilde{x}$ are distinct and contained in $\tilde{u}$, and no names in $\tilde{x}$ are fused by $\mathrm{Eq}(P')$. Similarly, the names $\tilde{z}$ are distinct and contained in $\tilde{v}$, and no names in $\tilde{z}$ are fused by $\mathrm{Eq}(Q')$. Therefore,

$$(\tilde{y}{=}\tilde{u}\backslash\tilde{x}) \oplus \mathrm{Eq}(P') \quad = \quad (\tilde{y}{=}\tilde{v}\backslash\tilde{z}) \oplus \mathrm{Eq}(Q'). \tag{4}$$

Now some of the names $\tilde{u}$ are bound by $\tilde{x}$, and some are not. Let the projection functions π_1 and π_2 project those names that are, and those that are not, respectively. That is to say, $\{\pi_1\tilde{u}\} \subseteq \{\tilde{x}\}$ and $\{\pi_2\tilde{u}\} \cap \{\tilde{x}\} = \emptyset$. Hence, every name in $\pi_1\tilde{u}$ is bound, and no name in $\pi_2\tilde{u}$ is. We can therefore rewrite the left side of Equation 4:

$$(\pi_1\tilde{y}{=}\pi_1\tilde{u})\backslash\tilde{x} \oplus \pi_2\tilde{y}{=}\pi_2\tilde{u} \oplus \mathrm{Eq}(P'). \tag{5}$$

This relates every $\pi_2\tilde{y}$ to some name not in $\tilde{y}$. Note that $(\pi_1\tilde{y}{=}\pi_1\tilde{u})\backslash\tilde{x}$ generates an equivalence relation which only relates names in $\pi_1\tilde{y}$ to each other. This is because all $\pi_1\tilde{u}$ were assumed to be in $\tilde{x}$. Then, since $\tilde{y}$ was assumed fresh and not in P', Equation 5 relates no name in $\pi_1\tilde{y}$ to any name not in $\tilde{y}$.

By Lemma 36, the right hand side of Equation 4 also relates every $\pi_2\tilde{y}$ to some name not in $\tilde{y}$, and no $\pi_1\tilde{y}$ to any name not in $\tilde{y}$. We can therefore rewrite the equation:

$$(\pi_1\tilde{y}{=}\pi_1\tilde{u})\backslash\tilde{x} \oplus \pi_2\tilde{y}{=}\pi_2\tilde{u} \oplus \mathrm{Eq}(P') \quad = \quad (\pi_1\tilde{y}{=}\pi_1\tilde{v})\backslash\tilde{z} \oplus \pi_2\tilde{y}{=}\pi_2\tilde{v} \oplus \mathrm{Eq}(Q'). \tag{6}$$

We can analyse each side of Equation 6 in two parts: the first part only relates names in $\pi_1\tilde{y}$ to each other, and the second relates names in $\pi_2\tilde{y}$ to various free names. The first part is as follows:

$$(\pi_1\tilde{y}{=}\pi_1\tilde{u})\backslash\tilde{x} \quad = \quad (\pi_1\tilde{y}{=}\pi_1\tilde{v})\backslash\tilde{z}. \tag{7}$$

Because $\{\pi_1\tilde{u}\} \subseteq \{\tilde{x}\}$ by construction of π, and because $\{\tilde{x}\} \subseteq \{\pi_1\tilde{u}\}$, we get $\{\tilde{x}\} = \{\pi_1\tilde{u}\}$.

Let n be the length of the list $\pi_1\tilde{y}$, which is also the length of $\pi_1\tilde{u}$ and $\pi_1\tilde{v}$. Let m be the length of $\tilde{x}$. Consider the surjective function $f : \{1 \ldots n\} \mapsto \{1 \ldots m\}$ such that $f(i) = j$ when $u_i = x_j$. This function is well-defined because each $x_j \in \tilde{x}$ is distinct, and $\{\pi_1\tilde{u}\} = \{\tilde{x}\}$. Now the left side of Equation 7 relates only those names $y_i{=}y_j$ such that $f(i) = f(j)$. By Lemma 36, so does the right side. Therefore the same surjective function applies. The first consequence of this is that $|\tilde{z}| = |\tilde{x}|$. Since the names in $\tilde{z}$ are distinct, there is a substitution σ such that $\sigma\tilde{z} = \tilde{x}$. The substitution amounts to alpha-renaming. The second consequence is that $\pi_1\sigma\tilde{v} = \pi_1\tilde{u}$. Thus, we have shown that the bound names in $\tilde{u}$ are the same as those in $\tilde{v}$, up to alpha-renaming.

Next, we consider the free names in $\tilde{u}$. The second parts in Equation 6 are as follows:

$$\pi_2\tilde{y}{=}\pi_2\tilde{u} \oplus \mathrm{Eq}(P') \quad = \quad \pi_2\tilde{y}{=}\pi_2\tilde{v} \oplus \mathrm{Eq}(Q').$$

Suppose that some y_i is related to some w on the left side. Then the left side also relates w to u_i. Therefore, by Lemma 36, so does the right side. But the right side also relates y_i to v_i. Therefore, it relates v_i to u_i. Hence, $\pi_2\tilde{u} = \pi_2\tilde{v}$ up to $\mathrm{Eq}(P')$, and $\mathrm{Eq}(P') = \mathrm{Eq}(Q')$.

The previous results satisfy the definition of structural congruence on concretions (Definition 11): $(\tilde{z})(\tilde{v} : Q') \equiv (\tilde{x})(\tilde{u} : Q'\sigma)$ for $\sigma = \{\tilde{x}/\tilde{z}\}$. Hence, $Q \xrightarrow{\overline{u}} (\tilde{x})(\tilde{u} : Q'\sigma)$.

We now need to show that $P' \mathcal{S} Q'\sigma$. We know that $P'' \mathcal{S} Q''$. Writing this out in full, and alpha-renaming the concretion with Q',

$$(\tilde{x})(\tilde{y}{=}\tilde{u} \mid w_1{=}w_2 \mid P') \quad \mathcal{S} \quad (\tilde{x})(\tilde{y}{=}\tilde{u} \mid w_1{=}w_2 \mid Q'\sigma). \tag{8}$$

We will use the fact that each $\tilde{y}, w_1, w_2$ is distinct and fresh, and can therefore all be removed without loss of information. To achieve this, first note that $\mathcal{S}$ is by definition closed with respect to contexts. Also recall that $\pi_1\tilde{u}$ contains those names in $\tilde{u}$ that are bound by $\tilde{x}$, and that $\pi_2\tilde{u}$ contains those names that are not. Now construct a context $(\tilde{y}w_1w_2)(\pi_1\tilde{y}{=}\pi_1\tilde{u} \mid \_)$, and apply it to both sides of Equation 8. Applying it to the left side yields the following.

$$\begin{array}{lll}
 & (\tilde{y}w_1w_2)\big(\pi_1\tilde{y}{=}\pi_1\tilde{u} \mid (\tilde{x})(\tilde{y}{=}\tilde{u} \mid w_1{=}w_2 \mid P')\big) & \\
\equiv & (\tilde{y})\big(\pi_1\tilde{y}{=}\pi_1\tilde{u} \mid (\tilde{x})(\tilde{y}{=}\tilde{u} \mid P')\big) & \text{remove } w_1, w_2 \\
\equiv & (\tilde{y})\big(\pi_1\tilde{y}{=}\pi_1\tilde{u} \mid \pi_2\tilde{y}{=}\pi_2\tilde{u} \mid (\tilde{x})(\pi_1\tilde{y}{=}\pi_1\tilde{u} \mid P')\big) & \text{partition } \tilde{y}{=}\tilde{u} \\
\equiv & (\tilde{y})\big(\pi_1\tilde{y}{=}\pi_1\tilde{u} \mid \pi_2\tilde{y}{=}\pi_2\tilde{u} \mid P'\{\pi_1\tilde{y}/\pi_1\tilde{u}\}\big) & \text{substitute, since } \{\tilde{x}\}{=}\{\pi_1\tilde{u}\} \\
\equiv & (\pi_2\tilde{y})\big(\pi_2\tilde{y}{=}\pi_2\tilde{u} \mid P'\big) & \text{partition } (\tilde{y}) \text{ and subst. back} \\
\equiv & P' & \text{since } \tilde{y} \text{ not in } P'
\end{array}$$

Similarly, applying the context to the right hand side yields $Q'\sigma$. From Equation 8 we get

$$P' \mathcal{S} Q'\sigma$$

as desired. Therefore, the largest barbed congruence is a ground bisimulation. □

Conclusions. This concludes our study of strong bisimulation for the explicit fusion calculus. We have seen that barbed congruence and ground congruence coincide in the strong case. We have also found an efficient characterisation for them.

Barbed congruence is an easy way to compare different calculi. In the following chapter we use it to relate the explicit fusion calculus to the pi calculus, and in Chapter 6 we use it to prove the fusion machine correct. The efficient bisimulation is a more useful proof technique for comparing one program to another: it is a sound and complete way to practically judge whether a given relation is a bisimulation congruence.

Before going on to Chapter 4, we consider weak bisimulation. Our results for weak bisimulation are incomplete. However, the rest of the dissertation does not depend on them.

3.9 Weak bisimulation

In *weak bisimulation*, we do not count the number of internal steps. This has a practical motivation: it lets us treat programs as 'black boxes', ignoring their internal working, observing only the commitments they can make.

However, it is harder to work with weak bisimulation than strong bisimulation, and there are pitfalls. Parrow and Victor proposed a congruence called *weak hyper-equivalence* for the fusion calculus [53], and claimed that that it is equal to weak barbed congruence. But Fu proved [23] that the two relations are in fact different. He has subsequently discovered [24] that, in the presence of mismatch, weak hyper-equivalence is not even a behavioural equivalence. However, the work of Fu and Parrow and Victor was done in the *finite* subcalculus—i.e. without the replication operator. We outline some properties of the full calculus, including replication.

It is possible to write a program which, through a number of internal steps, exchanges two names. In this sense (up to weak bisimulation) the program is like an explicit fusion. The simplest such program is an *equator*, first introduced by Honda and Yoshida [31]:

$$\mathcal{E}(u,v) \;=\; !u(\tilde{x}).\overline{v}\tilde{x} \mid !v(\tilde{x}).\overline{u}\tilde{x}$$

Here is an example execution of the equator:

$$\begin{aligned} & \mathcal{E}(u,v) \mid \overline{u}\tilde{w} \mid v\tilde{z} \\ \equiv\; & \mathcal{E}(u,v) \mid u(\tilde{x}).\overline{v}\tilde{x} \mid \overline{u}\tilde{w} \mid v\tilde{z} \\ \longrightarrow\; & \mathcal{E}(u,v) \mid \overline{v}\tilde{w} \mid v\tilde{z} \\ \longrightarrow\; & \mathcal{E}(u,v) \mid \tilde{w}{=}\tilde{z} \end{aligned}$$

As in this example, any rendezvous on u can be converted into one on v. Therefore, in the presence of something that interchanges names, the programs $\overline{u}\tilde{w}$ and $\overline{u}\tilde{z}$ are equivalent. So too are $\overline{x}u$ and $\overline{x}v$. However, the labelled transitions we used for strong ground bisimulation would judge these last two inequivalent, since they evolve to unequal interfaces:

$$\begin{aligned} \mathcal{E}(u,v) \mid \overline{x}v &\xrightarrow{\overline{x}} v : \mathcal{E}(u,v) \\ \mathcal{E}(u,v) \mid \overline{x}u &\xrightarrow{\overline{x}} u : \mathcal{E}(u,v) \end{aligned}$$

This means that we cannot use ground bisimulation for the weak case. We shall use barbed bisimulation $\dot{\approx}_b$ instead, since it seems a natural definition and since it does not suffer from the same problem (it does not use interfaces). Write $\xrightarrow{\tau}{}^*$ for a sequence of zero or more $\xrightarrow{\tau}$ transitions.

Definition 39 (Weak) *A weak barbed bisimulation is a symmetric relation $\mathcal{S}$ such that $P\ \mathcal{S}\ Q$ implies*

- *if $P \xrightarrow{\mu}$ then $Q \xrightarrow{\tau}{}^* \xrightarrow{\mu}$, and*
- *if $P \xrightarrow{\tau} P'$ then there exists a Q' such that $Q \xrightarrow{\tau}{}^* Q'$ and $P'\ \mathcal{S}\ Q'$*

It is also a weak barbed congruence when $P\ \mathcal{S}\ Q$ implies

- *for all contexts E, $E[P]\ \mathcal{S}\ E[Q]$.*

The largest weak barbed bisimulation $\dot{\approx}_b$ exists and is an equivalence. So too does the largest weak barbed congruence $\approx_b$.

The remainder of this section is dedicated to a proof about equators: namely, that $\mathcal{E}(u,v)$ is weakly bisimilar to $\mathcal{E}(u,v) \mid u{=}v$. We conjecture that in fact they are also barbed congruent. The significance of this conjecture would be that the inside-outside theorem (Theorem 18), key to relating strong ground and barbed congruence, fails in the weak case.

Since we will be using the equator $\mathcal{E}(u,v)$ and the substitution $\{u/v\}$ frequently, we abbreviate them in this section to just $\mathcal{E}$ and σ. We first call to attention some relevant aspects of equators. Equators act as *buffers*, storing for a time the data that was sent, and also losing information as to the order in which it was sent. Thus, in

$$\mathcal{E} \mid \overline{u}x.\overline{u}y.P \mid v(a).v(b).Q \quad \xrightarrow{\tau}\xrightarrow{\tau} \quad \mathcal{E} \mid \overline{v}x \mid \overline{v}y \mid P \mid v(a).v(b).Q,$$

the names x and y might be delivered in any order. By contrast, with an explicit fusion, the order is fixed:

$$u{=}v \mid \overline{u}x.\overline{u}y.P \mid v(a).v(b).Q \quad \xrightarrow{\tau}\xrightarrow{\tau} \quad u{=}v \mid P \mid Q\{x/a\}\{y/b\}.$$

Note also that, up to weak bisimulation, there is no effective difference between storing data in a u buffer or a v buffer: they can be interchanged, as shown below.

$$\mathcal{E} \mid \overline{u}x \quad \xrightarrow{\tau} \quad \mathcal{E} \mid \overline{v}x.$$

Although the propositions in this section concern barbed bisimulation (without interfaces), we nevertheless find it convenient to use interfaces in their proofs. We will use the abbreviations

$$P \stackrel{\mu}{\Longrightarrow} I : P' \text{ for } P \xrightarrow{\tau}{}^{*} \xrightarrow{\mu} J : P'',\ P'' \xrightarrow{\tau}{}^{*} P', \text{ and } J : P' \equiv I : P'$$
$$P \stackrel{\tau}{\Longrightarrow} P' \text{ for } P \xrightarrow{\tau}{}^{*} P'.$$

First, a minor lemma. It echoes Lemma 14 (page 36) for explicit fusions.

Lemma 40 *If $\mathcal{E} \mid P \xrightarrow{\lambda}_e P'$ then there exists a P'' such that $P' \equiv \mathcal{E} \mid P''$.*

Proof. A rule induction on the derivation of the transition. □

Proposition 41 $\mathcal{E}(u,v) \mid P \ \dot{\approx}_b \ \mathcal{E}(u,v) \mid P\{u/v\}$.

Proof. The proof of the proposition is substantial. In essence, it is just a large case analysis to construct the labelled transition graph of $\mathcal{E}(u,v) \mid P$ and that of $\mathcal{E}(u,v) \mid P\{u/v\}$, and showing that the graphs are the same up to weak bisimulation.

For the proof of the proposition, we will use structural induction on terms in the explicit fusion calculus. As always, the task is to find a good induction hypothesis. We will use a variant of ground bisimulation for this purpose, but modified to ignore the differences in interface. We use this because barbed bisimulation is too weak for the parallel case: it cannot give the transitions of $P|Q$

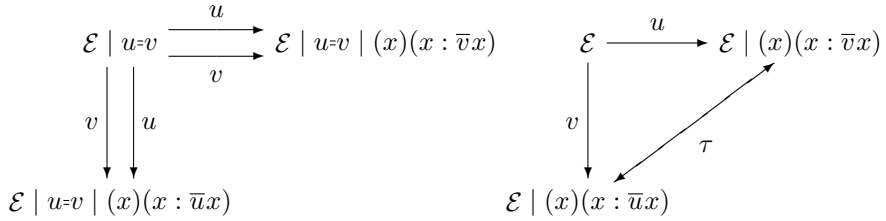
merely from those of P and Q separately. Our variant of ground bisimulation can give the transition of $P|Q$.

We will construct a relation $\mathcal{S}= \{(\mathcal{E}\,|P, \mathcal{E}\,|Q)\}$ where P and Q are identical up to u and v. The bulk of the proof is an induction to show that all such P and Q satisfy a particular property, such that the relation is a barbed congruence. We call the property *renamability*. Before defining renamability, we introduce some notation. Write $P\sigma = Q\sigma$ to mean that P and Q are structurally identical up to differences in u and v, and assuming alpha renaming. Write $(\tilde{x})(\tilde{y}_1 : P')\sigma = (\tilde{x})(\tilde{y}_2 : Q')\sigma$ to mean that $\tilde{y}_1\sigma = \tilde{y}_2\sigma$ and $P'\sigma = Q'\sigma$. Now a term P is renamable if for every Q such that $P\sigma = Q\sigma$,

- $\mathcal{E} \mid P \stackrel{\alpha}{\Longrightarrow} \mathcal{E} \mid I : P'$ implies $\mathcal{E} \mid Q \stackrel{\alpha}{\Longrightarrow} \mathcal{E} \mid J : Q'$ with $(I : P')\sigma = (J : Q')\sigma$ and P' renamable.

where we assume I does not bind $\mathcal{E}$.

The induction hypothesis allows a reaction $P \xrightarrow{\mu} I : P'$ to be matched by $Q \stackrel{\mu}{\Longrightarrow} I : Q'$, where Q may perform several tau transitions after it has made the commitment μ. We will use these extra tau transitions to relate the transitions of $\mathcal{E} \mid u{=}v$ to those of $\mathcal{E}$. The following commutative diagrams show this relation, showing how the transitions of $\mathcal{E} \mid u{=}v$ can be matched by $\mathcal{E}$ with a trailing tau. Note that they cannot be matched without that trailing tau.



These trailing tau transitions are purely an internal technique for the proof of the proposition. But the proposition itself is about barbed bisimulation, which ignores the results of these transitions.

Before proving that all terms are renamable, we first provide a more convenient characterisation of renamability.

Lemma 42 (Renamability) *The following conditions are sufficient for renamability:*

- $P \xrightarrow{\mu} I : P'$ *implies* $\mathcal{E} \mid Q \stackrel{\mu}{\Longrightarrow} \mathcal{E} \mid J : Q'$ *where* $(I : P')\sigma = (J : Q')\sigma$,
- $P \xrightarrow{\tau} P'$ *implies* $\mathcal{E} \mid Q \stackrel{\tau}{\Longrightarrow} \mathcal{E} \mid Q'$ *where* $P'\sigma = Q'\sigma$,
- *and each of the above implies* P' *to be renamable.*

Proof. Suppose $\mathcal{E} \mid P$ undergoes a $\xrightarrow{\alpha}$ transition. Recall that α ranges over μ and τ transitions. From Proposition 25, there are four main possibilities as to what this transition might have been.

1. $\mathcal{E} \mid P \xrightarrow{w} \mathcal{E} \mid (\tilde{x})(\tilde{x} : \overline{v}\tilde{x} \mid P)$, with $P \vdash u{=}w$. Because $P\sigma = Q\sigma$, then either $Q \vdash u{=}w$ or $Q \vdash v{=}w$. In the first case, we can deduce a translation

$$\mathcal{E} \mid Q \stackrel{u}{\Longrightarrow} \mathcal{E} \mid (\tilde{x})(\tilde{x} : \overline{v}\tilde{x} \mid Q)$$

from one part of the equator. We can then rename this transition to w. In the second case, we can deduce a similar transition from the other part of the equator, and also rename it to w. Both cases satisfy the definition of renamability.

2. $\mathcal{E} \mid P \xrightarrow{\mu} \mathcal{E} \mid I : P'$, with $P \xrightarrow{\mu} I : P'$. Renamability follows from the first condition in the lemma.

3. $\mathcal{E} \mid P \xrightarrow{\tau} \mathcal{E} \mid P'$, with $P \xrightarrow{\tau} P'$. Renamability follows from the second condition in the lemma.

4. $\mathcal{E} \mid P \xrightarrow{\tau} \mathcal{E} \mid (\tilde{y})(\overline{v}\tilde{z} \mid P')$, with $P \xrightarrow{\overline{u}} (\tilde{y})(\tilde{z} : P')$. From the second condition in the lemma,

$$\mathcal{E} \mid Q \stackrel{\overline{u}}{\Longrightarrow} \mathcal{E} \mid (\tilde{y})(\tilde{z} : Q')$$

where $(\tilde{y})(\tilde{z} : Q')$ and $(\tilde{y})(\tilde{z} : P')$ are equivalent up to $u{=}v$. Now we compose this with the term $u(\tilde{x}).\overline{v}\tilde{x}$. This allows us to deduce an internal reaction

$$u(\tilde{x}).\overline{v}\tilde{x} \mid \mathcal{E} \mid Q \stackrel{\tau}{\Longrightarrow} \mathcal{E} \mid (\tilde{y})(\overline{v}\tilde{z} \mid Q').$$

However, $u(\tilde{x}).\overline{v}\tilde{x} \mid \mathcal{E}$ is structurally congruent to $\mathcal{E}$. This yields the desired result, satisfying renamability:

$$\mathcal{E} \mid Q \stackrel{\tau}{\Longrightarrow} \mathcal{E} \mid (\tilde{y})(\overline{v}\tilde{z} \mid Q'). \qquad \square$$

We now use this lemma in proving that every term is renamable. We do this by induction. This continues the proof of Proposition 41.

Null, fusion. The terms $\mathbf{0}$ and ϕ are renamable, since neither has any transitions.

Prefix. The term $\mu\tilde{y}.P$ is renamable, as follows. The possibilities are that μ is u, v, $\overline{u}$, $\overline{v}$ or some $w \notin \{u, v\}$. We will consider all $\mu_1\tilde{y}_1.P_1$ which are identical to $\mu\tilde{y}.P$ up to u and v.

1. Suppose $\mu = u$. Then either $\mu_1 = u$, in which case $\mu_1\tilde{y}_1.P_1$ undergoes the required $\xrightarrow{u}$ transition immediately. Or $\mu_1 = v$. In this case, the equator can make the $\xrightarrow{v}$ commitment and then perform a tau step to update the resulting buffer:

$$\begin{aligned}\mathcal{E} \mid v\tilde{y}_1.P_1 &\xrightarrow{u} \mathcal{E} \mid (\tilde{x})(\tilde{x} : \overline{v}\tilde{x} \mid v\tilde{y}_1.P_1) \\ &\xrightarrow{\tau} \mathcal{E} \mid \tilde{y}_1 : P_1\end{aligned}$$

   The mirror case where $\mu = v$ is similar.

2. Suppose $\mu = \overline{u}$. Then either $\mu_1 = \overline{u}$, in which case $\mu_1\tilde{y}_1.P_1$ undergoes the required $\xrightarrow{\overline{u}}$ transition immediately. Or $\mu_1 = \overline{v}$. In this case, μ_1 can make an internal reaction with the equator first, and then commit on $\xrightarrow{\overline{u}}$:

$$\begin{aligned}\mathcal{E} \mid \overline{v}\tilde{y}_1.P_1 &\xrightarrow{\tau} \mathcal{E} \mid \overline{u}\tilde{y}_1 \mid P_1 \\ &\xrightarrow{\overline{u}} \mathcal{E} \mid yy_1 : P_1\end{aligned}$$

   The mirror case where $\mu = \overline{u}$ is similar.

3. Suppose $\mu \notin \{u, v, \overline{u}, \overline{v}\}$. Then $\mu_1 = \mu$, and so $\mu_1 \tilde{y}_1.P_1$ undergoes the required $\xrightarrow{\mu}$ transition immediately.

Restriction. If P is renamable, then so is $(z)P$, as follows. Suppose $(z)P \xrightarrow{\mu} C$. Then by Proposition 25, there exists I and P' such that $C = (z)(I : P')$ and $P \xrightarrow{\mu} I : P'$ with $z \notin \mu$. By the induction hypothesis, $\mathcal{E} \mid Q \stackrel{\mu}{\Longrightarrow} \mathcal{E} \mid J : Q'$ for any Q equal to P up to u and v, such that $I : P'$ and $J : Q'$ are also equal. Assume by alpha-renaming that $z \notin \{u, v\}$. We can directly deduce $\mathcal{E} \mid (z)Q \stackrel{\mu}{\Longrightarrow} \mathcal{E} \mid (z)J : Q'$.

Parallel. If P and Q are renamable, then so is $P \mid Q$, as follows. We use Proposition 25 to analyse the possible transitions of $P \mid Q$.

1. The first case is that $P \mid Q$ undergoes a $\xrightarrow{\mu'}$ transition, which originates either from P or Q. Suppose P: that is,

$$P \xrightarrow{\mu} I : P' \text{ and } P|Q \vdash \mu' = \mu,$$
$$\text{yielding } P \mid Q \xrightarrow{\mu} I : P' \mid Q.$$

   We will prove that $\mathcal{E} \mid P_1 \mid Q_1 \stackrel{\mu'}{\Longrightarrow} \mathcal{E} \mid J : P_1' \mid Q_1$ for any $P_1 \mid Q_1$ equal to $P \mid Q$ up to u and v. Because it is so equal, $\mathrm{Eq}(u{=}v \mid P \mid Q) = \mathrm{Eq}(u{=}v \mid P_1 \mid Q_1)$. Therefore either $P_1|Q_1 \vdash \mu{=}\mu'$ or (supposing μ to be an input) $P_1|Q_1 \vdash \mu{=}u, v{=}\mu'$, or some equivalent symmetrical case.

   Now from the induction hypothesis, $\mathcal{E} \mid P_1 \stackrel{\mu}{\Longrightarrow} \mathcal{E} \mid J : P_1'$. Therefore $\mathcal{E} \mid P_1 \mid Q_1 \stackrel{\mu}{\Longrightarrow} \mathcal{E} \mid J : P_1' \mid Q_1$. In the case where $P_1|Q_1 \vdash \mu{=}\mu'$, we can transform this directly into

$$\mathcal{E} \mid P_1 \mid Q_1 \stackrel{\mu'}{\Longrightarrow} \mathcal{E} \mid J : P_1' \mid Q_1$$

   as desired. Otherwise, it depends on whether μ is an input or an output commitment.

   If μ is an input, we will add a term $v(\tilde{x}).\overline{u}\tilde{x}$ which can perform a $\xrightarrow{v}$ transition. In the presence of $P_1|Q_1$ we can rename v to μ'. Supposing that $J = (\tilde{y})(\tilde{z} : \_)$ we get

$$v(\tilde{x}).\overline{u}\tilde{x} \mid \mathcal{E} \mid P_1 \mid Q_1 \xrightarrow{\mu'} (\tilde{x})(\tilde{x} : \overline{u}\tilde{x} \mid \mathcal{E} \mid P_1 \mid Q_1). \tag{9}$$

   We are given that $P_1|Q_1 \vdash u{=}\mu$. Now the buffer can do an output on u, and the term $\mathcal{E} \mid P_1 \mid Q_1$ can do an input on μ, so the two can react:

$$\overline{u}\tilde{x} \mid \mathcal{E} \mid P_1 \mid Q_1 \stackrel{\tau}{\Longrightarrow} \mathcal{E} \mid (\tilde{y})(\tilde{z}{=}\tilde{x} \mid P_1' \mid Q_1). \tag{10}$$

   Using Equation 10 as a sequence of tau steps at the end of Equation 9 we get the desired result:

$$\begin{aligned} v(\tilde{x}).\overline{u}\tilde{x} \mid \mathcal{E} \mid P_1 \mid Q_1 &\stackrel{\mu'}{\Longrightarrow} (\tilde{x})(\tilde{x} : \mathcal{E} \mid (\tilde{y})(\tilde{z}{=}\tilde{x} \mid P_1 \mid Q_1)) \\ &\equiv \mathcal{E} \mid (\tilde{y})(\tilde{z} : P_1' \mid Q_1). \end{aligned}$$

If μ is an output, we have the reverse situation. Compose the term $u(\tilde{x}).\overline{v}\tilde{x}$. Because $P_1|Q_1 \vdash u{=}\mu$, there can be an internal $\stackrel{\tau}{\Longrightarrow}$ transition, leaving a buffer $\overline{v}\tilde{x}$. Now rename this buffer to $\overline{\mu'}\tilde{x}$, and the result can do the appropriate μ' output transition.

2. The second parallel case is that $P \mid Q$ undergoes an internal reaction due to either P or Q alone. By the induction hypothesis, either $\mathcal{E} \mid P_1$ or $\mathcal{E} \mid Q_1$ does as well.

3. The final parallel case is that $P \mid Q$ undergoes an internal reaction due to an output from one and an input from the other. We consider the case of

$$P \stackrel{\overline{x}}{\longrightarrow} I : P', \quad Q \stackrel{y}{\longrightarrow} J : Q',$$
$$P|Q \vdash x{=}y$$

yielding $P \mid Q \stackrel{\tau}{\longrightarrow} I@J[P' \mid Q']$. By the induction hypothesis,

$$\mathcal{E} \mid P_1 \stackrel{\overline{x}}{\Longrightarrow} \mathcal{E} \mid I_1 : P_1', \quad \mathcal{E} \mid Q_1 \stackrel{y}{\Longrightarrow} \mathcal{E} \mid J_1 : Q_1',$$
$$\text{either } P_1|Q_1 \vdash x{=}y \text{ or } P_1|Q_1 \vdash x{=}u, v{=}y$$

If the first case is true, then we get

$$\mathcal{E} \mid \mathcal{E} \mid P_1 \mid Q_1 \stackrel{\tau}{\Longrightarrow} \mathcal{E} \mid \mathcal{E} \mid I_1@J_1[P_1' \mid Q_1'].$$

Now any derivation $\mathcal{E} \mid R \stackrel{\alpha}{\Longrightarrow} \mathcal{E} \mid R'$ must be finite. It can therefore have used only a finite number of copies of the equator. Therefore there exist terms E and E', each finite copies, such that $E \mid R \stackrel{\alpha}{\Longrightarrow} E' \mid R'$. Moreover, $\mathcal{E} \equiv E \mid \mathcal{E} \equiv E' \mid \mathcal{E}$. Applying this general principle to the equation, we get

$$\mathcal{E} \mid P_1 \mid Q_1 \stackrel{\tau}{\Longrightarrow} \mathcal{E} \mid I_1@J_1[P_1' \mid Q_1']$$

as desired. Finally, I_1 and J_1 are related to I and J through σ, and so $I_1@J_1[P_1' \mid Q_1']$ and $I@J[P' \mid Q']$ are equal up to u and v.

If the second case is true, then we must use an instance of the equator. Let ϕ be $\mathrm{Eq}(Q_1)$ and ψ be $\mathrm{Eq}(P_1)$. Then, from the induction hypothesis, and assuming $I_1 = (\tilde{y})(\tilde{z} : \_)$, we make the following chain of deductions:

$$\begin{aligned}
\mathcal{E} \mid P_1 \mid \phi &\stackrel{\overline{x}}{\Longrightarrow} \mathcal{E} \mid I_1 : P_1' \mid \phi && \text{from induction hypothesis} \\
\mathcal{E} \mid P_1 \mid \phi &\stackrel{\overline{u}}{\Longrightarrow} \mathcal{E} \mid I_1 : P_1' \mid \phi && \text{since } x = u \\
u(\tilde{x}).\overline{v}\tilde{x} \mid \mathcal{E} \mid P_1 \mid \phi &\stackrel{\overline{u}}{\Longrightarrow} \mathcal{E} \mid (\tilde{y})(\overline{v}\tilde{z} \mid P_1' \mid \phi) && \text{composition instance of } \mathcal{E} \\
\mathcal{E} \mid P_1 \mid \phi &\stackrel{\overline{u}}{\Longrightarrow} \mathcal{E} \mid (\tilde{y})(\overline{v}\tilde{z} \mid P_1' \mid \phi) && \text{use } \equiv \text{ to remove duplicate}
\end{aligned}$$

Also from the induction hypothesis, and renaming the label of the transition, $\mathcal{E} \mid \psi \mid Q_1 \stackrel{v}{\Longrightarrow} \mathcal{E} \mid \psi \mid J_1 : Q_1'$. Reacting the two together, and removing the second copy of the equator as before,

$$\mathcal{E} \mid P_1 \mid Q_1 \mid \phi \mid \psi \stackrel{\tau}{\Longrightarrow} \mathcal{E} \mid I_1@J_1[P_1' \mid Q_1' \mid \phi \mid \psi].$$

And since ϕ and ψ are merely some explicit fusions already contained in the term, we can get rid of them:

$$\mathcal{E} \mid P_1 \mid Q_1 \stackrel{\tau}{\Longrightarrow} \mathcal{E} \mid I_1@J_1[P_1' \mid Q_1'].$$

Replication If P is renamable, then so is $!P$. This case is almost identical to the parallel case.

This concludes the proof that all terms P are renamable. We can now conclude the proof of Proposition 41. Consider the relation $\mathcal{S}= \{(\mathcal{E}\,|P, \mathcal{E}\,|Q)\}$ where P and Q are identical up to u and v. Since barbed bisimulation is a special case of renamability, and all P, Q are renamable, $\mathcal{S}$ is a barbed bisimulation. □

Corollary 43 $\mathcal{E} \dot{\approx}_b \mathcal{E} \mid u{=}v$.

Conjecture 44 $\mathcal{E}(u, v) \approx_b u{=}v \mid \mathcal{E}(u, v)$.

The impact of this conjecture is that would demonstrate two terms that are congruent even though they have different explicit fusions. Such a situation is not possible in the strong case (Theorem 18). Therefore, in the strong case, we can syntactically determine whether two names are interchangeable (i.e. whether $P \sim_b P \mid u{=}v$). But if the conjecture holds, then such a syntactic determination is not possible in the weak case. For consider a modified equator $!u(x).H.\overline{v}x \mid !v(x).\overline{u}x$, where H encodes one of the standard undecidable problems in computer science. Either H terminates, making this program behave like an equator. Or H does not terminate, and u and v are not interchangeable. Now if we had some characterisation of which names are interchangeable, then we could decide whether H terminates. Therefore, such a characterisation is impossible. Without such a characterisation, there is no way to define weak ground labels in such a way as to make weak ground congruence coincide with weak barbed congruence.

We have shown (Proposition 41) that equators allow names to be interchanged in all parallel contexts. As discussed above, it remains to seen whether they allow interchange in all contexts.

Chapter 4

Embedding into explicit fusions

The fusion calculus and the pi calculus embed naturally into the explicit fusion calculus. This chapter proves that their embeddings are correct. For each calculus, I first recall its own definition of bisimulation, and then establish the connection between this and bisimulation for the explicit fusion calculus. The plan of the chapter is as follows.

4.1 *Overview.* We explain what an embedding is, and what it means for an embedding to be correct.

4.2 *The fusion calculus.* We recall the fusion calculus [52] and its definition of *hyper-equivalence* (the name for its version of bisimulation). The fusion calculus uses a different style of labelled transitions.

4.3 *Embedding the fusion calculus.* Consider two terms in the fusion calculus. They are hyper-equivalent in the fusion calculus if and only if they are congruent in the explicit fusion calculus: the embedding is sound and complete ('fully abstract').

4.4 *The pi calculus.* We recall the pi calculus and its definition of barbed bisimulation.

4.5 *Embedding the pi calculus.* Consider pi calculus contexts translated into the explicit fusion calculus. If two pi programs are barbed bisimilar in all pi contexts, then their translations are barbed bisimilar in all translated contexts: the embedding is sound.

4.1 Overview

This chapter concerns embeddings from one calculus into another. We now state mathematically what it means for an embedding to be correct. We then explain and motivate this statement with an analogy.

A calculus X has a set $\mathcal{P}_X$ of terms and an equivalence relation $\sim_X$ over terms. An embedding from one calculus X into another Y is a translation $(\cdot)^* :$

$\mathcal{P}_X \mapsto \mathcal{P}_Y$. An embedding might have several properties, described below. Let P_X range over $\mathcal{P}_X$.

- A *sound* embedding is one where $P_X \sim_X Q_X$ implies $P_X^* \sim_Y Q_X^*$.
- A *complete* embedding is one where $P_X^* \sim_Y Q_X^*$ implies $P_X \sim_X Q_X$.
- A *fully abstract* embedding is one that is both sound and complete.

As we have seen, there are actually a collection of different equivalences $\sim_X$ for each calculus. Rather than just talking about an embedding of X into Y, we should strictly talk about an embedding of X with some particular $\sim_X$ into Y with some particular $\sim_Y$.

Since our calculi are all concurrent calculi, and they all have similar definitions of equivalence, we often like to find that more properties are preserved by embedding. Suppose that our calculus X has a parallel composition operator $|_X$: $\mathcal{P}_X \times \mathcal{P}_X \mapsto \mathcal{P}_X$, a transition relation $\longrightarrow_X \subseteq \mathcal{P}_X \times \mathcal{P}_X$, an observation relation $\stackrel{\cdot}{\longrightarrow}_X \subseteq \mathcal{P}_X \times (\mathcal{N} \cup \overline{\mathcal{N}})$, and a set of contexts $\mathcal{E}_X : \mathcal{P}_X \mapsto \mathcal{P}_X$. We are interested in the following properties.

- A *compositional* embedding is one where the translation of a term is defined with respect to its sub-terms: for example, $(P_X \mid_X Q_X)^* = P_X^* \mid_Y Q_X^*$.
- We say that *reaction is preserved* when $P_X \longrightarrow_X Q_X$ implies $P_X^* \longrightarrow_Y Q_X^*$, and also $P_X^* \longrightarrow_Y Q_Y$ implies $\exists Q_X : Q_Y = Q_X^* \Rightarrow P_X \longrightarrow_X Q_X$. We say that reaction is *strongly* preserved when it is preserved, and also $P_X^* \longrightarrow_Y Q_Y$ implies $\exists Q_X : Q_Y = Q_X^* \wedge P_X \longrightarrow Q_X$.
- If observations are strongly preserved ($P_X \stackrel{\mu}{\longrightarrow}_X \Leftrightarrow P_X^* \stackrel{\mu}{\longrightarrow}_Y$) and reaction is strongly preserved, then the embedding itself has the properties of a barbed bisimulation. (We cannot actually say that the embedding *is* a bisimulation, because bisimulation was only defined within a single calculus X to be a subset of $\mathcal{P}_X \times \mathcal{P}_X$, and the embedding is a subset of $\mathcal{P}_X \times \mathcal{P}_Y$.) Note that if the embedding has the properties of a barbed bisimulation, then it also preserves barbed bisimulation.
- An embedding is *sound with respect to the contexts* $\mathcal{E}'_Y \subseteq \mathcal{E}_Y$, when $\forall E_X : E_X[P_X] \sim_X E_X[Q_X]$ implies $\forall E_Y \in \mathcal{E}'_Y : E_Y[P_X^*] \sim_X E_Y[Q_X^*]$. Similarly, an embedding can be complete and fully abstract with respect to contexts $\mathcal{E}'_Y$.

To understand what the correctness properties mean, and which ones are important, I find it helpful to think of a story from Simak's science fiction novel *City* [65]. Humans have reached Mars, but the environment there is too inhospitable. They build a machine to transmogrify themselves into better-adapted martian creatures. In martian form they lose some of their human senses, and they acquire new martian senses. In our analogy, humans are terms in calculus X, martians are terms in calculus Y, their sensible attributes are observation and reaction, and transmogrification is translation.

Soundness with respect to translated contexts. The first important property is that indistinguishability is retained by transmogrification, from the perspective

of other transmogrified people. Thus, if no humans could tell a particular pair of human twins apart, then no transmogrified humans can tell the transmogrified twins apart either. In the context of an implementation this is an important practical property. If humans are programs in a language, and martians are their implementation, then this property means that two library routines deemed equivalent by the language specification will also be equivalent when executed (so long as all other executing programs are also written in that language). In the context of the pi calculus we prove that if two programs are barbed bisimilar in all pi contexts, then their translations into the explicit fusion context will also be barbed bisimilar in all 'piable' explicit fusion contexts. (Piability identifies the image of pi contexts under translation.)

Full abstraction. The second important property is whether all martians (both native martians and humans after transmogrification) are as discriminating as native humans. If everyone on Mars perceives the same similarities and differences in people as all humans do, then the social structures in martian society will be exactly the same as those on Earth. In the context of the fusion calculus, we prove that the structure of hyper-equivalence in the fusion calculus is the same as the structure of congruence in the explicit fusion calculus. In the context of the fusion machine, we prove that machine-equivalence is the same as shallow barbed congruence in the explicit fusion calculus.

Embedding has properties of barbed bisimulation. The third property, for the special case where humans and martians have the same sensory apparatus, is that transmogrification does not alter any of an object's sensible attributes. This means that a human will appear the same after transmogrification as before: no one else, neither human nor martian, can tell them apart. This is a practically useful property for programmers: it means that when they inspect how their program runs at the machine level (martian), they will find that it uses the same messages and names as they programmed into it at the calculus level (human). We also prove this property for our fragmentation: a fragmented term is observationally indistinguishable from its original.

4.2 The fusion calculus recalled

This section is a brief summary of the fusion calculus and its bisimulation. It largely restates existing work of Victor and Parrow [52, 69]. This is in preparation for the following section, which proves full abstraction between the fusion calculus and the explicit fusion calculus.

The fusion calculus has a very different labelled transition system from the explicit fusion calculus. This section therefore goes into some depth explaining the fusion transition system, and provides lemmas relating it to the explicit fusion transition system. It is instructive to compare the definition of fusion transitions (Definition 47, page 63) with that of explicit fusion transitions (Definition 13, page 35), and also fusion reaction (Definition 46, page 62) with explicit fusion reaction (Definition 4, page 24). Explicit fusions lead to a simpler presentation of both.

It is interesting that, despite their very different labelled transition systems and definitions of bisimulation, the fusion calculus and the explicit fusion calculus should have exactly the same bisimulation congruence. This leads us to suspect that the bisimulation congruence is a natural one for these calculi. Note

that although the calculi have the same congruence, there remains an important difference between them: the fusion calculus has a non-local reaction relation making it awkward to implement, while the explicit fusion calculus has an easier local reaction relation.

Definition 45 (Fusion calculus) *The set $\mathcal{P}_{\text{fu}}$ of fusion terms is given by*

$$P ::= \mathbf{0} \quad | \quad P|P \quad | \quad !P \quad | \quad (x)P \quad | \quad \overline{u}\tilde{x}.P \quad | \quad u\tilde{x}.P.$$

The structural congruence between terms, $\equiv$, is as for the explicit fusion calculus (Definition 3, page 23), but without the rules for fusion interchange.

Definition 46 (Fusion reaction) *The reaction relation $\longrightarrow_{\text{fu}}$ is the smallest relation satisfying the following rule, and closed with respect to $\equiv$,*

$$(\tilde{z})(\overline{u}\tilde{x}.P \mid u\tilde{y}.Q \mid R) \quad \longrightarrow_{\text{fu}} \quad P\sigma \mid Q\sigma \mid R\sigma,$$

where σ is a substitution satisfying the following properties: $\text{ran}(\sigma), \text{dom}(\sigma) \subseteq \{\tilde{x}, \tilde{y}\}$*; $\tilde{z} = \text{dom}(\sigma) \backslash \text{ran}(\sigma)$; and $\sigma(v) = \sigma(w)$ iff $(v, w) \in \text{Eq}(\tilde{x}{=}\tilde{y})$, where $\text{Eq}(\tilde{x}{=}\tilde{y})$ is the smallest equivalence relation containing each (x_i, y_i).*

This definition can perhaps be explained more easily through its connection to the explicit fusion calculus:

- If $P \longrightarrow Q$ in the explicit fusion calculus, and neither P nor Q contain any explicit fusions, then $P \longrightarrow_{\text{fu}} Q$ in the fusion calculus.
- If $P \longrightarrow_{\text{fu}} Q$, then $P \longrightarrow Q$.

This connection is just a special case of full abstraction, proved in the following section.

We now move to the transition relation in the fusion calculus. We will define the labels and the transition first, and then explain them. Recall that μ ranges over $\mathcal{N} \cup \overline{\mathcal{N}}$. Let δ range over *non-binding labels* $\{\mu\tilde{x}, !\tilde{x}{=}\tilde{y}\}$, and let ϵ range over *possibly binding labels* $\{\delta, (\tilde{y})\mu\tilde{x}\}$ where $\tilde{y} \subseteq \tilde{x}$ and $\mu \notin \tilde{y}$. The free and bound names of the various labels are given by

$$\begin{aligned}
\text{fn}(u) &= u \\
\text{fn}(\overline{u}) &= u \\
\text{fn}(\mu\tilde{x}) &= \text{fn}(\mu) \cup \{\tilde{x}\} \\
\text{fn}(!\tilde{x}{=}\tilde{y}) &= \{u : \exists v : (u, v) \in \text{Eq}(\tilde{x}{=}\tilde{y}),\ u \neq v\} \\
\text{fn}(\ (\tilde{y})\overline{u}\tilde{x}\ ) &= \{\tilde{x}u\} \backslash \tilde{y} \\
\text{fn}(\ (\tilde{y})u\tilde{x}\ ) &= \{\tilde{x}u\} \backslash \tilde{y} \\
\text{bn}((\tilde{y})\mu\tilde{x}) &= \{\tilde{y}\}
\end{aligned}$$

Note that the free names of a fusion label are all those names that are fused not only to themselves. This is different from the explicit fusion calculus, where the the name u is free in $u{=}u$. This is not a significant difference—just an awkward one.

Definition 47 (Fusion transitions) *The labelled transition system for the fusion calculus, $P \xrightarrow{\epsilon}_{\text{fu}} P'$, is given by*

$$\overline{u}\tilde{x}.P \xrightarrow{\overline{u}\tilde{x}}_{\text{fu}} P \qquad u\tilde{x}.P \xrightarrow{u\tilde{x}}_{\text{fu}} P$$

$$\frac{P \xrightarrow{\overline{u}\tilde{x}}_{\text{fu}} P' \quad Q \xrightarrow{u\tilde{y}}_{\text{fu}} Q'}{P \mid Q \xrightarrow{!\tilde{x}=\tilde{y}}_{\text{fu}} P' \mid Q'} \qquad \frac{P \xrightarrow{\delta}_{\text{fu}} P'}{P \mid Q \xrightarrow{\delta}_{\text{fu}} P' \mid Q}$$

$$\frac{P \xrightarrow{\delta}_{\text{fu}} P' \quad x \notin \text{fn}(\delta)}{(x)P \xrightarrow{\delta}_{\text{fu}} (x)P'} \qquad \frac{P \xrightarrow{!\tilde{x}=\tilde{y}}_{\text{fu}} P' \quad (u,v) \in \text{Eq}(\tilde{x}=\tilde{y}) \quad u \neq v}{(u)P \xrightarrow{!\tilde{x}=\tilde{y}\setminus u}_{\text{fu}} P'\{v/u\}}$$

$$\frac{P_1 \equiv P \xrightarrow{\delta}_{\text{fu}} Q \equiv Q_1}{P_1 \xrightarrow{\delta}_{\text{fu}} Q_1} \qquad \frac{P \xrightarrow{(\tilde{y})\mu\tilde{z}}_{\text{fu}} P' \quad x \in \tilde{z} - \tilde{y} \quad x \notin \mu}{(x)P \xrightarrow{(x\tilde{y})\mu\tilde{z}}_{\text{fu}} P'}$$

We briefly explain the labelled transition system. The fusion calculus uses a *'tell' fusion transition* to indicate that an internal reaction has caused a fusion: for example,

$$\overline{u}x.P \mid uy.Q \xrightarrow{!x=y}_{\text{fu}} P \mid Q.$$

This fusion has its fusing effect during reaction, if a reaction ends up being allowed. It has potentially global effect, up to some delimiting restriction. Therefore the transition can only be discharged in the presence of that restriction:

$$\frac{\overline{u}x.P \mid uy.Q \mid R \xrightarrow{!x=y}_{\text{fu}} P \mid Q \mid R}{(x)(\overline{u}x.P \mid uy.Q \mid R) \xrightarrow{\tau}_{\text{fu}} (P \mid Q \mid R)\{y/x\}}$$

We sometimes write $P \xrightarrow{!\phi} \text{fu}P'$ to indicate that the transition causes a fusion, but without having to specify which names are fused. The identity fusion transition $P \xrightarrow{!\mathbf{I}}_{\text{fu}} P'$ has no fusing effect, and is equivalent to the conventional tau transition.

Recall that the reaction relation in the fusion calculus only allows reaction if all fusion have been discharged through restriction. In effect, the 'tell' fusion label indicates that there is a potential reaction, but only if enough restrictions are present to remove fusions. This makes an interesting parallel with the 'ask' fusion label in the explicit fusion calculus: this label indicates a potential reaction, but only if enough explicit fusions are provided (Section 3.6).

The fusion calculus distinguishes between possibly binding labels ϵ and non-binding labels δ. Note that the rules for output, input, parallel composition and structural congruence apply only to non-binding labels. To deduce a binding labelled transition $P \mid (x)\overline{u}x.Q \xrightarrow{(x)\overline{u}x}_{\text{fu}} P \mid Q$ it is necessary to first move the restriction to the outside using scope-extrusion, then deduce a transition from the contents $P \mid \overline{u}x.Q$, and finally re-apply the restriction. This procedure is used in Lemma 50 to deduce some derived transition rules that are closer in spirit to those of the explicit fusion calculus.

In order to describe a behavioural congruence, the fusion calculus uses the notion of a *substitutive effect* of a fusion label. This effect is a substitution that

sends all members of each equivalence class to one representative of the class. The substitutive effect of all other labels is just the identity substitution.

Definition 48 (Substitutive effect) *A substitution σ agrees with the fusion ϕ if $\forall x, y.\ (x, y) \in \phi \Leftrightarrow \sigma(x){=}\sigma(y)$. A substitutive effect of a fusion ϕ is a substitution σ agreeing with ϕ such that $\forall x, y.\ \sigma(x){=}y \Rightarrow x\phi y$.*

Definition 49 (Hyper-equivalence) *A hyper bisimulation is a symmetric relation $\mathcal{S}$ such that whenever $P\ \mathcal{S}\ Q$, then for all substitutions σ*

- *if $P\sigma \xrightarrow{\epsilon} P'$ with $\mathrm{bn}(\epsilon) \cap \mathrm{fn}(Q\sigma) = \emptyset$ then $Q\sigma \xrightarrow{\lambda} Q'$ and $P'\rho\ \mathcal{S}\ Q'\rho$, for some substitutive effect ρ of ϵ.*

Two terms P and Q are hyper-equivalent*, written $P \sim_{\mathrm{fu}} Q$, if and only if there exists a hyper-bisimulation between them. The relation $\sim_{\mathrm{fu}}$ is the largest hyper-bisimulation.*

Note that the original fusion calculus paper defines bisimulation and hyper-equivalence separately, while my restatement has combined them for convenience.

In the following lemma I deduce some derived transition rules for the fusion calculus. These derived transitions are closer in spirit to those of the explicit fusion calculus. In particular, they give closure properties for binding labels $(\tilde{y})\mu\tilde{x}$; Definition 47 only gives closure properties for non-binding labels $\mu\tilde{x}$. We give this lemma in preparation for the following section, where it is used to prove the connection between fusion labels and explicit fusion labels.

Lemma 50

1. *$P \xrightarrow{(\tilde{y})\mu\tilde{x}}_{\mathrm{fu}} P'$ implies $(z)P \xrightarrow{(\tilde{y})\mu\tilde{x}}_{\mathrm{fu}} (z)P'$ if $z \notin \{\mu, \tilde{x}, \tilde{y}\}$.*
2. *$P \xrightarrow{(\tilde{y})\mu\tilde{x}}_{\mathrm{fu}} P'$ implies $P \mid Q \xrightarrow{(\tilde{y})\mu\tilde{x}}_{\mathrm{fu}} P' \mid Q$ assuming $\tilde{y} \notin \mathrm{fn}(Q)$.*
3. *$P \xrightarrow{(\tilde{y})\mu\tilde{x}}_{\mathrm{fu}} P'$ implies $!P \xrightarrow{(\tilde{y})\mu\tilde{x}}_{\mathrm{fu}} P' \mid !P$ assuming $\tilde{y} \notin \mathrm{fn}(!P)$.*
4. *$P \xrightarrow{(\tilde{y}_1)\mu\tilde{x}_1}_{\mathrm{fu}} P'$ and $Q \xrightarrow{(\tilde{y}_2)\overline{\mu}\tilde{x}_2}_{\mathrm{fu}} Q'$ imply $P \mid Q \xrightarrow{!(\tilde{x}_1{=}\tilde{x}_2)\setminus\tilde{y}_a}_{\mathrm{fu}} (\tilde{y}_b)(P'\sigma \mid Q'\sigma)$, with the following side-conditions. The names $\tilde{y}_a$ and $\tilde{y}_b$ are distinct and partition $\{\tilde{y}_1, \tilde{y}_2\}$. Each name in $\tilde{y}_a$ is fused with an element not in $\tilde{y}_a$ through the fusion $\tilde{x}_1{=}\tilde{x}_2$, and σ substitutes each element in $\tilde{y}_a$ accordingly. The names $\tilde{y}_b$ are not affected by the fusion $(\tilde{x}_1{=}\tilde{x}_2)\setminus\tilde{y}_a$. We assume no clashes: $\tilde{y}_1$ and $\tilde{y}_2$ are distinct, and $\{\tilde{y}_1\} \cap \mathrm{fn}(Q) = \{\tilde{y}_2\} \cap \mathrm{fn}(P) = \emptyset$.*

The final derived rule is awkward to state formally. Essentially, as many of the binders $\tilde{y}_1, \tilde{y}_2$ as possible are removed by interchanging them with other names from the fusion $\tilde{x}_1{=}\tilde{x}_2$: these constitute $\tilde{y}_a$. And $\tilde{y}_b$ contains those names that cannot be removed.

4.3 Full abstraction for fusion calculus

We now prove that the explicit fusion calculus is a fully abstract model of the fusion calculus, with respect to strong congruence. We have already discussed

the significance of this result: it reassures us that we have identified the 'correct' congruence, and allows us to use the simpler definitions of equivalence defined in the explicit fusion calculus, rather than those defined in the fusion calculus. In effect, the fusion calculus can be subsumed within the explicit fusion calculus.

It is awkward to prove full abstraction for the fusion calculus using barbed congruence. This is because the fusion calculus and the explicit fusion calculus have different reaction relations. Instead, we will relate hyper-equivalence in the fusion calculus to ground bisimulation congruence in the explicit fusion calculus. A connection between barbed congruence in the two calculi follows directly: we have shown in Chapter 3 that ground congruence is equal to barbed congruence in the explicit fusion calculus, and meanwhile Victor and Parrow have shown [70] that hyper-equivalence is equal to barbed congruence in the fusion calculus.

First, a remark about notation. The set $\mathcal{P}_{\text{fu}}$ of terms in the fusion calculus is a subset of the terms $\mathcal{P}_\phi$ in the explicit fusion calculus. When we write a fusion transition $P \xrightarrow{\epsilon}_{\text{fu}} P'$ it is apparent that both P and P' are in $\mathcal{P}_{\text{fu}}$. When we write an explicit fusion transition, we will use a term with an asterisk P^* to range over terms in $\mathcal{P}_{\text{fu}}$, and P to range over all of $\mathcal{P}_\phi$.

We now consider the connection between labels in the fusion calculus (Definition 47, page 63) and labels in the explicit fusion calculus (Definition 13, page 35). The fusion calculus labels are different in two ways. First, the 'tell' transition $P \xrightarrow{!x=y}_{\text{fu}} P'$ of the fusion calculus *tells* the environment that, if reaction is allowed, then it will cause a fusion. In contrast, the 'ask' fusion transition $P \xrightarrow{?x=y} P'$ of the explicit fusion calculus *asks* for an explicit fusion to be present in order to allow reaction. Second, the fusion calculus carries the data to be communicated in the label $P \xrightarrow{\overline{u}x}_{\text{fu}} P'$; the explicit fusion calculus carries it in a concretion $P \xrightarrow{\overline{u}} x : P'$.

Despite these differences, there is a straightforward connection between the two labelled transition systems: apart from the channel name itself, all other information conveyed in a fusion calculus label (including the 'tell' fusion label) is conveyed in the explicit fusion calculus by the interface and explicit fusions of the resulting concretion; and the 'ask' fusion labels in the explicit fusion calculus are not actually needed—they are merely present as a convenience, to allow an efficient characterisation (Section 3.5), and can be deduced from the other labels. The connection between the labels, stated formally in the following lemma, is illustrated in the table below:

Fusion calculus		*Explicit fusion calculus*
$\overline{u}x.P \xrightarrow{\overline{u}x}_{\text{fu}} P$	...	$\overline{u}x.P^* \xrightarrow{\overline{u}}_e x : P^*$
$(x)\overline{u}x.P \xrightarrow{(x)\overline{u}x}_{\text{fu}} P$	...	$(x)\overline{u}x.P^* \xrightarrow{\overline{u}}_e (x)(x : P^*)$
$\overline{u}x.P \mid uy.Q \xrightarrow{!x=y}_{\text{fu}} P \mid Q$	...	$\overline{u}x.P^* \mid uy.Q^* \xrightarrow{\tau}_e x{=}y \mid P^* \mid Q^*$

Lemma 51 *Let P be a term in $\mathcal{P}_{\text{fu}}$. Then P^* has no explicit fusions. Furthermore,*

1. *if P undergoes a transition in the fusion calculus, the transition is one of*

   (a) *$P \xrightarrow{(\tilde{y})u\tilde{x}}_{\text{fu}} P'$ such that $P^* \xrightarrow{u}_e (\tilde{y})(\tilde{x} : P'^*)$, or similarly for $\xrightarrow{(\tilde{y})\overline{u}\tilde{x}}_{\text{fu}}$*

(b) $P \xrightarrow{!\phi}_{\mathrm{fu}} P'$ *such that* $P^* \xrightarrow{\tau}_e \phi \mid P'^*$;

2. *if* P^* *undergoes a transition in the explicit fusion calculus, it is one of*

(a) $P^* \xrightarrow{u}_e \equiv (\tilde{y})(\tilde{x} : P_1'^*)$ *such that* $P \xrightarrow{(\tilde{y})u\tilde{x}}_{\mathrm{fu}} P_1'$, *or similarly for* $\xrightarrow{\overline{u}}_e$

(b) $P^* \xrightarrow{?x=y}_e \equiv \phi \mid P_1'^*$ *such that* $\exists P_2'.\ P\{y/x\} \xrightarrow{!\phi}_{\mathrm{fu}} P_2'$, *and* $x=y \mid \phi \mid P_1'^* \equiv x=y \mid \phi \mid P_2'^*$

(c) $P^* \xrightarrow{\tau}_e \equiv \phi \mid P_1'^*$ *such that* $\exists P_2'.\ P \xrightarrow{!\phi}_{\mathrm{fu}} P_2'$, *and* $\phi \mid P_1'^* \equiv \phi \mid P_2'^*$.

Proof. The fact that P^* has no explicit fusions follows from the definition of $(\cdot)^*$. Part 1 of the lemma is proved by induction on the derivation of the transition $\xrightarrow{\epsilon}_{\mathrm{fu}}$. Part 2 is proved by induction on the structure of P (which is also the structure of P^*) using Lemma 50. Part 2b looks complicated because the explicit fusion ϕ in $P \xrightarrow{?x=y}_e \phi \mid P_1'^*$ can have immediate effect on $P_1'^*$, but the fusion label ϕ in the transition $P \xrightarrow{!\phi}_{\mathrm{fu}} P_2'$ only has effect after the process has been enclosed by a restriction. □

We now prove the connection between hyper-equivalence in the fusion calculus, and efficient bisimulation in the explicit fusion calculus. In essence, given a hyper-equivalence that relates P to Q, the corresponding efficient bisimulation will relate $\phi|P$ to $\phi|Q$ for all explicit fusions ϕ.

There are two interesting parts to the proof. The first is our reconstruction of an 'ask' fusion transition in the explicit fusion calculus, from a tau transition in the fusion calculus. This reconstruction illustrates the point made above—that 'ask' transitions are not fundamental, but merely a convenience to avoid having to quantify over all contexts. The second is our reconstruction of a 'tell' fusion transition from a tau transition in the explicit fusion calculus. This illustrates another point made above—that in the fusion calculus the labels carry the data, but in the explicit fusion calculus this data is carried in the interface and explicit fusions of the resulting term.

Theorem 52 $P \sim_{\mathrm{fu}} Q$ *if and only if* $P^* \overset{\mathrm{e}}{\sim}_g Q^*$.

Proof. In the forwards direction, we construct a relation $\mathcal{S}$ on explicit fusion terms such that $P\ \mathcal{S}\ Q$ iff $P \equiv (\tilde{x})(\phi \mid P_1^*)$ and $Q \equiv (\tilde{x})(\phi \mid Q_1^*)$ such that $P_1 \sim_{\mathrm{fu}} Q_1$. We prove that $\mathcal{S}$ is an efficient congruence. Clearly the explicit fusions match. As for the transitions, Proposition 25 says that any transition of $(\tilde{x})(\phi|Q_1^*)$ must have been deduced from a corresponding transition of $\phi|Q_1^*$. Therefore we consider the transitions of $P\ \mathcal{S}\ Q$ such that $P \equiv \phi \mid P_1^*, Q \equiv \phi \mid Q_1^*$ and $P_1 \sim_{\mathrm{fu}} Q_1$. Since P_1^* and Q_1^* have no explicit fusions, it follows that $\mathrm{Eq}(P) = \mathrm{Eq}(Q)$. This leaves two parts of efficient congruence definition 21 to satisfy. We will use Lemmas 51 and 25 to characterise their possible transitions.

1. First consider the transition

$$\phi \mid P_1^* \equiv P \xrightarrow{\overline{u}}_e \equiv I : P' \quad \text{with} \quad P_1^* \xrightarrow{\overline{v}}_e I : P_1'^*,\ P' \equiv \phi \mid P_1'^*,\ \phi \vdash u=v,$$

where $I = (\tilde{x})(\tilde{y} : \_)$ and $\tilde{x}$ does not bind ϕ. Then $P_1 \xrightarrow{(\tilde{x})\overline{v}\tilde{y}}_{\mathrm{fu}} P_1'$. Since $P_1 \sim_{\mathrm{fu}} Q_1$, we obtain $Q_1 \xrightarrow{(\tilde{x})\overline{v}\tilde{y}}_{\mathrm{fu}} Q_1'$ with $P_1' \sim_{\mathrm{fu}} Q_1'$. By Lemma 51,

$Q_1^* \xrightarrow{\overline{v}}_e I : Q_1'^*$ and hence $\phi \mid Q_1^* \xrightarrow{\overline{u}}_{\pi_F} I : (\phi \mid Q_1'^*)$. Finally, $\phi \mid P_1'^* \; \mathcal{S}$ $\phi \mid Q_1'^*$ by construction of $\mathcal{S}$. An analogous result holds for the input case.

2. Now consider the transition

$$\phi \mid P_1^* \equiv P \xrightarrow{?x=y}_e P' \quad \text{with} \quad P_1^* \xrightarrow{?u=v}_e \psi | P_1'^*,\ P' \equiv \phi|\psi|P_1'^*,\ \phi \vdash uv{=}xy.$$

Given the transition $P_1^* \xrightarrow{?u=v}_e \psi \mid P_1'^*$ we need to reconstruct the fact that Q_1^* can undergo a τ-transition: writing ρ for a substitutive effect of ψ,

$$\begin{aligned} P_1^* \xrightarrow{?u=v}_e \psi|P_1'^* &\Rightarrow \exists P_2'.\ P_1\{u/v\} \xrightarrow{!\psi}_{\text{fu}} P_2' \text{ with } u{=}v|\psi|P_1'^* \equiv u{=}v|\psi|P_2'^* \\ &\Rightarrow Q_1\{u/v\} \xrightarrow{!\psi}_{\text{fu}} Q_2' \text{ with } P_2'\rho \sim_{\text{fu}} Q_2'\rho \\ &\Rightarrow Q_1^*\{u/v\} \xrightarrow{\tau}_e \psi \mid Q_2'^* \text{ with } (P_2'\rho)^* \; \mathcal{S} \; (Q_2'\rho)^* \\ &\Rightarrow u{=}v \mid \phi \mid Q_1^* \xrightarrow{\tau}_e u{=}v \mid \phi \mid \psi \mid Q_2'^* \end{aligned}$$

Finally, $P_2'\rho^* \; \mathcal{S} \; Q_2'\rho^*$ implies $\psi \mid P_2'^* \; \mathcal{S} \; \psi \mid Q_2'^*$. From the closure properties of $\mathcal{S}$, and since $u{=}v \mid P' \equiv u{=}v \mid \phi \mid \psi \mid P_2'^*$, we fulfill the requirement that $u{=}v \mid P' \; \mathcal{S} \; u{=}v \mid \phi \mid \psi \mid Q_2'^*$. An analogous result holds for the tau transition.

In the reverse direction, we construct a relation $\mathcal{S}$ on fusion terms such that $P \; \mathcal{S} \; Q$ iff $P^* \overset{\text{e}}{\sim}_g Q^*$. It remains to prove that the relation $\mathcal{S}$ is a hyper-equivalence. Note that $\mathcal{S}$ is closed with respect to substitution, since the substitution $\{y/x\}$ can be expressed as the context $(x)(x{=}y \mid \_)$, $\overset{\text{e}}{\sim}_g$ is closed with respect to all contexts (Theorem 30, page 44), and $(\_)^*$ preserves substitution. It is therefore enough to analyse a label $P \xrightarrow{\lambda}_{\text{fu}} P'$ without substitution, since all substitutions $P\sigma \xrightarrow{\lambda}_{\text{fu}} P''$ automatically follow. Lemma 51 accounts for output and input labels. For a 'tell' fusion label, suppose that $P \xrightarrow{!\phi}_{\text{fu}} P'$. From Lemmas 51 and 25,

$$\begin{aligned} P \xrightarrow{!\phi}_{\text{fu}} P' &\Rightarrow P^* \xrightarrow{\tau}_e \phi \mid P'^* \\ &\Rightarrow Q^* \xrightarrow{\tau}_e \phi \mid Q'^* \text{ with } \phi \mid P'^* \overset{\text{e}}{\sim}_g \phi \mid Q'^*. \end{aligned}$$

From Lemma 51 we see that Q has a corresponding transition $Q \xrightarrow{!\phi}_{\text{fu}} Q_1'$ with $\phi \mid Q_1'^* \equiv \phi \mid Q'^*$. Applying an appropriate restriction context to $\phi \mid P'^* \overset{\text{e}}{\sim}_g \phi \mid Q'^*$, we get the desired substitutive effect ρ of ϕ: that is, $P'^*\rho \overset{\text{e}}{\sim}_g Q'^*\rho$, and hence $P'\rho \; \mathcal{S} \; Q_1'\rho$. □

Note that this section concerns only strong bisimulation congruences. The results do not extend to the weak case. Indeed, Fu has shown that weak hyper-equivalence in the fusion calculus is not even a congruence [23].

4.4 The pi calculus recalled

This section is a concise definition of the pi calculus and its barbed bisimulation. It merely restates existing work; more detailed accounts may be found in recent

books by Milner [45] and Sangiorgi and Walker [60]. We adopt the same set $\mathcal{N}$ of names as for the explicit fusion calculus (Section 2.2, page 23), and the same definitions of bound and free names.

Definition 53 (Pi calculus) *The set $\mathcal{P}_\pi$ of pi terms and $\mathcal{E}_\pi$ of pi contexts are given by*

$$\begin{array}{lcl} P & ::= & \boldsymbol{0} \mid P|P \mid !P \mid (x)P \mid \overline{u}\tilde{x}.P \mid u(\tilde{x}).P \\ E & ::= & \_ \mid \overline{u}\tilde{x}.E \mid u(\tilde{x}).E \mid !E \mid (x)E \mid P|E \mid E|P \end{array}$$

where, in $u(\tilde{x})$, the $\tilde{x}$ are distinct.

Structural congruence for the pi calculus is like that for the explicit fusion calculus, but with alpha-renaming instead of fusion interchange.

Definition 54 (Pi structural congruence) *The structural congruence $\equiv$ between terms is the smallest equivalence relation satisfying the following axioms, and closed with respect to the contexts $\_ \mid \_$, $!\_$, $(\_)\_$ and $a\tilde{x}.\_$:*

1. *Abelian monoid laws with $\boldsymbol{0}$ as identity*
   $P \mid \boldsymbol{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
2. *scope laws*
   $(xy)P \equiv (yx)P \qquad (x)(P \mid Q) \equiv (x)P \mid Q$ *if* $x \notin \mathrm{fn}(P)$
3. *replication*
   $!P \equiv P \mid !P$
4. *alpha renaming*
   $(x)P \equiv (y)P\{y/x\} \qquad u(x).P \equiv u(y).P\{y/x\} \qquad$ *if* $y \notin \mathrm{fn}(P)$

Definition 55 (Pi observation, reaction) *The observation relation $P \xrightarrow{\mu}$ is the smallest relation satisfying*

$$\overline{u}\tilde{x}.P \xrightarrow{\overline{u}}$$
$$u(\tilde{x}).P \xrightarrow{u}$$

$$P \mid Q \xrightarrow{\mu} \quad \text{if } P \xrightarrow{\mu}$$
$$(x)P \xrightarrow{\mu} \quad \text{if } P \xrightarrow{\mu} \text{ and } x \notin \mu Q \qquad\qquad \xrightarrow{\mu} \quad \text{if } Q \equiv P \xrightarrow{\mu}$$

The internal reaction relation $P \xrightarrow{\tau} P'$ is the smallest relation satisfying

$$\overline{u}\tilde{x}.P \mid u(\tilde{y}).Q \xrightarrow{\tau} (P|Q)\{\tilde{x}/\tilde{y}\}$$

$$P \mid Q \xrightarrow{\tau} P' \mid Q \quad \text{if } P \xrightarrow{\tau} P'$$
$$(x)P \xrightarrow{\tau} (x)P' \quad \text{if } P \xrightarrow{\tau} P'$$
$$Q \xrightarrow{\tau} Q' \quad \text{if } Q \equiv P \xrightarrow{\tau} P' \equiv Q'$$

Barbed bisimulation and shallow barbed congruence are defined for the pi calculus in the same way as for the explicit fusion calculus (Definitions 33 and 35, page 33).

4.5 Embedding the pi calculus

In this section we define the translation from the pi calculus into the explicit fusion calculus, and prove that it is sound with respect to shallow barbed congruence on translated contexts.

Definition 56 *The translation* $(\_)^* : \mathcal{P}_\pi \to \mathcal{P}_\phi$ *is as follows.*

$$\begin{aligned}
\boldsymbol{0}^* &= \boldsymbol{0} \\
(P \mid Q)^* &= P^* \mid Q^* \\
(!P)^* &= !(P^*) \\
((x)P)^* &= (x)(P^*) \\
(\overline{u}\tilde{x}.P)^* &= \overline{u}\tilde{x}.(P^*) \\
(u(\tilde{x}).P)^* &= (\tilde{x}')(u\tilde{x}'.(P\{\tilde{x}'/\tilde{x}\}^*)), \text{ distinct } \tilde{x}' \notin \{u\} \cup \operatorname{fn} P
\end{aligned}$$

The final line of the translation uses distinct fresh names $\tilde{x}'$ to account for the possibility that $u \in \tilde{x}$. For consider the pi calculus term $x(x).P$. It cannot be translated as $(x)xx.P^*$, since that would incorrectly bind the subject of the action. By using substitution on the right hand side, rather than just assuming that $u \notin \tilde{x}$, we obtain a translation that is purely structural on the left hand side and does not need to be quotiented by alpha-renaming.

Definition 57 (Piability) *A term P_ϕ in the explicit fusion calculus is piable when there exists a term Q_π in the pi calculus such that $Q_\pi^* \equiv P_\phi$. A context E_ϕ in the explicit fusion calculus is piable when there exists a context E_π in the pi calculus such that for all Q_π, $E_\pi[Q_\pi]^* \equiv E_\phi[Q_\pi^*]$.*

Piability is a key concept in embedding the pi calculus into the explicit fusion calculus. We will see, for instance, that the set of piable contexts are no more discriminating than pi contexts. In Chapter 6 we extend the definition of piability to also talk about piable fusion machines.

We need to find an inductive characterisation of piability for use in formal proofs. First, consider some examples.

1. $(x)(\overline{u}x.Q_\pi^*)$ is piable: it is the image of $(x)\overline{u}x.Q_\pi$ under $(\cdot)^*$.
2. $(x)(ux.Q_\pi^*)$ is piable: it is the image of $u(x).Q_\pi$.
3. $(x)(x{=}y \mid \overline{u}x.Q_\pi^* \mid \overline{u}y)$ is piable: it is the image of $\overline{u}y.Q_\pi\{y/x\} \mid \overline{u}y$
4. $x{=}y \mid Q_\pi^*$ is not piable.
5. $(x)!(x{=}y \mid Q_\pi^*)$ is not piable.
6. $(x)(x{=}y \mid ux.Q_\pi^* \mid \overline{u}y)$ is not piable.
7. $(x)(xx.Q_\pi^*)$ is not piable.

The first three examples show the three roles played by restriction in the explicit fusion calculus. Sometimes, as in Example 1, a restriction is just a restriction. Sometimes, as in Example 2, it performs the binding in a bound input. Sometimes, as in Example 3, it discharges an explicit fusion. (As in Example 4, all fusions must be discharged.) Restriction cannot fulfill more than one role at a

time. For instance, in Example 6, it cannot both discharge the fusion and bind the input.

Example 7 shows a technical complication to do with alpha-renaming. The bound input $x(x).P$ in the pi calculus can only be translated as $(x')xx'.P^*$, where the bound input name does not clash with the subject of the communication.

In giving an inductive characterisation of piability, we will write pairs of the form $(\tilde{x}, P)$, where $\tilde{x}$ is a set of distinct names and P is a term in the explicit fusion calculus. This pair corresponds to the term $(\tilde{x})P$, but where every name in $\tilde{x}$ will play the role of binding an input within P. We will write relations of the form

$$(x, ux.Q^*_\pi) \; \mathcal{T} \; u(x).Q_\pi$$

to relate an explicit fusion term on the left to a pi calculus term on the right. The relation is a syntactic one, not quotiented by structural congruence or alpha-renaming. The three roles for restriction will be manifest in the rule for restriction. The relation $\mathcal{T}$ between pairs and terms in the pi calculus is as follows:

$$\begin{array}{rll}
(\emptyset, \mathbf{0}) \; \mathcal{T} \; \mathbf{0} & & \\
(\emptyset, \overline{u}\tilde{x}.P) \; \mathcal{T} \; \overline{u}\tilde{x}.Q & \text{if} & (\emptyset, P) \; \mathcal{T} \; Q \\
(\tilde{x}, u\tilde{x}.P) \; \mathcal{T} \; u(\tilde{x}).Q & \text{if} & (\emptyset, P) \; \mathcal{T} \; Q, u \notin \{\tilde{x}\} \\
(\emptyset, !P) \; \mathcal{T} \; !Q & \text{if} & (\emptyset, P) \; \mathcal{T} \; Q \\
(\tilde{x}, P_1|P_2) \; \mathcal{T} \; Q_1|Q_2 & \text{if} & \exists \tilde{x}_1, \tilde{x}_2 \text{ partitioning } \tilde{x} : \tilde{x}_1 \notin \mathrm{fn}(P_2), \; \tilde{x}_2 \notin \mathrm{fn}(P_1), \\
& & (\tilde{x}_1, P_1) \; \mathcal{T} \; Q_1, \; (\tilde{x}_2, P_2) \; \mathcal{T} \; Q_2 \\
(\tilde{x}, (z)P) \; \mathcal{T} \; Q & \text{if} & z \notin \{\tilde{x}\} \text{ and one of} \\
& & \begin{cases} (\tilde{x} \cup \{z\}, P) \; \mathcal{T} \; Q, \text{ or} \\ (\tilde{x}, P) \; \mathcal{T} \; Q' \text{ where } Q \text{ is } (z)Q', \text{ or} \\ \exists y : (\tilde{x}, P\{y/z\}) \; \mathcal{T} \; Q \text{ with } (z, y) \in \mathrm{Eq}(P), \; z \neq y. \end{cases}
\end{array}$$

We illustrate $\mathcal{T}$ by showing how to derive a relation for the explicit fusion term $(x)(vx.\mathbf{0} \mid (x)ux.\mathbf{0})$.

$$\begin{array}{rcll}
(x, ux.\mathbf{0}) & \mathcal{T} & u(x).\mathbf{0} & \text{from } \textit{in},\textit{nil} \quad (11) \\
(\emptyset, (x)ux.\mathbf{0}) & \mathcal{T} & u(x).\mathbf{0} & \text{from } \textit{res1},11 \quad (12) \\
 & & & \\
(x, vx.\mathbf{0}) & \mathcal{T} & v(x).\mathbf{0} & \text{from } \textit{in},\textit{nil} \quad (13) \\
(x, vx.\mathbf{0} \mid (x)ux.\mathbf{0}) & \mathcal{T} & v(x).\mathbf{0} \mid u(x).\mathbf{0} & \text{from } 12,13 \quad (14) \\
(\,\emptyset, (x)(vx.\mathbf{0} \mid (x)ux.\mathbf{0})\,) & \mathcal{T} & v(x).\mathbf{0} \mid u(x).\mathbf{0} & \text{from } \textit{res1},14 \quad (15)
\end{array}$$

The rule for restriction is subtle. It means, for instance, that none of the following three examples are related by $\mathcal{T}$ to any term in the pi calculus:

1. $(x, (x)ux.Q^*_\pi)$,
2. $(x, (x)Q^*_\pi)$,
3. $(x, (x)(y)(x{=}y \mid uy.Q^*_\pi)$.

The principle behind rejecting these examples is that in a pair $(\tilde{x}, P)$, every name in $\tilde{x}$ will be used as a bound input. In the examples, some outer restriction (x) had already been mistakenly placed in the first element of the pair, in the mistaken belief that it would be used for bound input. Whereas, in fact, the outer restriction can bind input in none of the examples.

Although we defined the relation $\mathcal{T}$ without alpha renaming, we can deduce alpha renaming:

Lemma 58

1. *If* $(\emptyset, P)\ \mathcal{T}\ Q$ *then* $(\emptyset, P\{y/x\})\ \mathcal{T}\ Q\{y/x\}$.
2. *If* $(\tilde{x}, P)\ \mathcal{T}\ Q$ *and* $\tilde{y} \notin \{\tilde{x}\} \cup \operatorname{fn} P$ *then* $(\tilde{y}, P\{\tilde{y}/\tilde{x}\})\ \mathcal{T}\ Q'$ *where* $Q' = Q$ *up to alpha renaming.*

Proof. Both parts are inductions on the derivation of the $\mathcal{T}$ relation. The first part is trivial, so I focus on the second. The principle is straightforward: given the derivation of $(\tilde{x}, P)\ \mathcal{T}\ Q$, there must be an almost identical derivation which uses $\tilde{y}$ instead of $\tilde{x}$. The proof is not difficult, just fiddly. I will write $(\tilde{x}, P)\ \mathcal{T}{=}\ Q$ as an abbreviation for $\exists Q' : (\tilde{x}, P)\ \mathcal{T}\ Q'$ and $Q = Q'$ up to alpha renaming. There are five ways that $(\tilde{x}, P)\ \mathcal{T}\ Q$ might have been deduced:

1. $(\tilde{x}, u\tilde{x}.P)\ \mathcal{T}\ u(\tilde{x}).Q$ with $(\emptyset, P)\ \mathcal{T}\ Q$ and $u \notin \tilde{x}$. By the first part, $(0, P\{\tilde{y}/\tilde{x}\})\ \mathcal{T}\ Q\{\tilde{y}/\tilde{x}\}$. Since $\tilde{y}$ does not clash with $u\tilde{x}.P$, $u \notin \tilde{y}$. Therefore $(\tilde{y}, u\tilde{y}.P\{\tilde{y}/\tilde{x}\})\ \mathcal{T}\ u(\tilde{y}).Q\{\tilde{y}/\tilde{x}\}$ as desired.
2. $(\tilde{x}, P_1|P_2)\ \mathcal{T}\ Q_1|Q_2$ with $\tilde{x}_1, P_1)\ \mathcal{T}\ Q_1$ and $(\tilde{x}_2, P_2)\ \mathcal{T}\ Q_2$ and $\tilde{x}_1 \notin \operatorname{fn} P_2$ and $\tilde{x}_2 \notin \operatorname{fn} P_1$. By the induction hypothesis, $(\tilde{y}_i, P_i\{\tilde{y}_i/\tilde{x}_i\})\ \mathcal{T}{=}\ Q_i$ for $i \in \{0, 1\}$. Now $\tilde{y}$ was chosen to be fresh; therefore $\tilde{y}_1 \notin \operatorname{fn} P_2\{\tilde{y}_2/\tilde{x}_2\}$ and $\tilde{y}_2 \notin \operatorname{fn} P_1\{\tilde{y}_1/\tilde{x}_1\}$. Therefore $(\tilde{y}, (P_1|P_2)\{\tilde{y}/\tilde{x}\})\ \mathcal{T}{=}\ Q_1|Q_2$ as desired.

The final three ways all have the form $(\tilde{x}, (z)P)\ \mathcal{T}\ Q$ with $z \notin \tilde{x}$. Given the compound substitution $\{\tilde{y}/\tilde{x}\} = \{y_1/x_1\} \ldots \{y_n/x_n\}$, we will one individual substitution $\{y/x\}$ at a time, for y being some y_i and x some x_i. Let $\sigma = \{y/x\}$. Now the substitution $((z)P)\{y/x\}$ has three possible forms according to whether (a) $x \neq z, y \neq z$ or (b) $x \neq z, y = z$ or (c) $x = z$. In fact the (c) form is not possible, since $(\tilde{x}, (z)P)$ can only have been deduced from $z \notin \tilde{x}$.

3a. $(\tilde{x}, (z)P)\ \mathcal{T}\ Q$ with $z \notin \tilde{x}$ and $(\tilde{x}z, P)\ \mathcal{T}\ Q)$ and $x \neq z$ and $y \neq z$. From the induction hypothesis, $(\tilde{x}z\sigma, P\sigma)\ \mathcal{T}{=}\ Q$. Since σ does not affect z we get $\tilde{x}z\sigma = \tilde{x}\sigma z$. And because $z \notin \tilde{x}\sigma$ we get $(\tilde{x}\sigma, (z)(P\sigma))\ \mathcal{T}{=}\ Q$ as desired.

3b. $(\tilde{x}, (z)P)\ \mathcal{T}\ Q$ with $z \notin \tilde{x}$ and $(\tilde{x}z, P)\ \mathcal{T}\ Q$. Now the substitution on $((z)P)\{y/x\}$ yields $(z')(P\{z'/z\}\{y/x\})$ for some z' not in P. Since x binds an input in P, $z \neq x$. Therefore $(\tilde{x}\sigma, P\{z'/z\}\{z/x\})\ \mathcal{T}{=}\ Q$, giving $(\tilde{x}\sigma, (z')(P\{z'/z\}\{z/x\}))\ \mathcal{T}{=}\ Q$ as desired. As shown, the (b) cases are not fundamentally different from the (a) cases, so we omit them in the following.

4a. $(\tilde{x}, (z)P)\ \mathcal{T}\ Q$ with $z \notin \tilde{x}$ and $y \neq z$ and $y \notin \operatorname{fn} P$ and $(\tilde{x}, P)\ \mathcal{T}\ Q$. By the induction hypothesis, $(\tilde{x}\sigma, P\sigma)\ \mathcal{T}{=}\ Q$. And since $z \notin \tilde{x}\sigma$, we get $(\tilde{x}\sigma, (z)(P\sigma))\ \mathcal{T}{=}\ (z)Q$ as desired.

5a. $(\tilde{x}, (z)P)\ \mathcal{T}\ Q$ with $z \notin \tilde{x}$ and $y \neq z$ and $y \notin \mathrm{fn}\, P$ and there exists a w such that $(\tilde{x}, P\{w/z\})\ \mathcal{T}\ Q$ and $(z, w) \in \mathrm{Eq}(P)$ and $z \neq w$. From the induction hypothesis, $(\tilde{x}\sigma, P\{w/z\}\{y/x\})\ \mathcal{T}{=}\ Q$. Now there are two further possibilities:

(a) $w = x$. Then the above equation becomes $(\tilde{x}\sigma, P\{y/x\}\{y/z\})\ \mathcal{T}{=}\ Q$ with $(z, x) \in \mathrm{Eq}(P)$ and $z \neq x$. Therefore $(z, y) \in \mathrm{Eq}(P\{y/x\})$. Moreover, since y is different from both x and z, we can swap the substitutions. This yields $(\tilde{x}\sigma, P\{y/x\}\{y/z\})\ \mathcal{T}{=}\ Q$, which leads directly to the desired result.

(b) $w \neq x$. Then we can re-order the substitutions from our induction hypothesis. This yields $(\tilde{x}\sigma, P\{y/x\}\{w/z\})\ \mathcal{T}{=}\ Q$, which leads directly to the desired result. □

The following lemmas lead to a statement that the relation $\mathcal{T}$ does indeed describe piability. In these lemmas we adopt the convention that P ranges over terms in the explicit fusion calculus and Q over terms in the pi calculus. The statements of the lemmas include subscripts P_ϕ and Q_π, but the subscripts are omitted within proofs.

Lemma 59 *$(\tilde{x}, P_\phi)\ \mathcal{T}\ Q_\pi$ implies $(\tilde{x})P_\phi \equiv Q_\pi^*$.*

Proof. A straightforward induction on the derivation of the left hand side (which amounts to an induction on the structure of P). □

Lemma 60 *Given Q_π, there exists a Q'_π such that $(0, Q_\pi^*)\ \mathcal{T}\ Q'_\pi$.*

Proof. An induction on the depth of structure of Q_π. We do not obtain the simpler statement $(0, Q_\pi^*)\ \mathcal{T}\ Q_\pi$ because the translation $(\cdot)^*$ renames the subject of bound input. For instance, if $Q_\pi = u(x).P$ then $Q_\pi^* = (x')(ux'.P\{x'/x\}$, and so we only get $(0, Q_\pi^*)\ \mathcal{T}\ u(x').P\{x'/x\})$. This issue is manifest in the induction step for bound input, which we now consider.

1. Now the induction goes over the depth of the structure of Q, and since $Q\{\tilde{x}'/\tilde{x}\}$ has lesser depth, the induction hypothesis yields

$$(\emptyset, Q\{\tilde{x}'/\tilde{x}\}^*)\ \mathcal{T}\ Q'$$

for some Q'. It follows that

$$(\tilde{x}', u\tilde{x}'.(Q\{\tilde{x}'/\tilde{x}\}^*))\ \mathcal{T}\ Q'.$$

Hence we get the desired result:

$$(\emptyset, (\tilde{x}')u\tilde{x}'.(Q\{\tilde{x}'/\tilde{x}\}^*))\ \mathcal{T}\ Q'$$

The other induction steps are straightforward. □

Lemma 61 *If $P_\phi \equiv P'_\phi$ and $\exists Q_\pi, \tilde{w} : (\tilde{w}, P_\phi)\ \mathcal{T}\ Q_\pi$ then $\exists Q'_\pi : Q'_\pi \equiv P'_\phi \wedge (\tilde{w}, P'_\phi)\ \mathcal{T}\ Q'_\pi$.*

Proof. The proof of the lemma is a lengthy induction over the derivation of $P \equiv P'$.

1. The Abelian monoid laws are straightforward. For instance, if $(\emptyset, P \mid Q)\ \mathcal{T}\ R'$, this must have been deduced from $(\emptyset, P)\ \mathcal{T}\ P'$ and $(\emptyset, Q)\ \mathcal{T}\ Q'$ and $R' = P' \mid Q'$. Therefore, changing the order, $(\emptyset, Q \mid P)\ \mathcal{T}\ Q' \mid P'$.

2. The first scope law, $(xy)P \equiv (yx)P$, expands out to nine different cases, according to which of the three restriction roles was played by x, and then which of the three was played by y. In the first case, for instance, both x and y are used to bind an input within P, so that we deduced $(\emptyset, (xy)P)\ \mathcal{T}\ P'$ from $(xy, P)\ \mathcal{T}\ P'$. We can easily reconstruct $(y, (x)P)\ \mathcal{T}\ P'$ and then $(\emptyset, (yx)P)\ \mathcal{T}\ P'$. Another case is when x was used to bind input, and y was used to discharge a fusion. This would happen in the example

$$(\emptyset, (xy)(y{=}x \mid ux.P^*))\ \mathcal{T}\ u(x).P\{x/y\}. \tag{16}$$

   In this case, the relation is deduced from $(x, (y)(y{=}x \mid ux.P^*))\ \mathcal{T}\ R'$, which in turn is deduced from $(x, (y{=}x \mid ux.P^*)\{x/y\})\ \mathcal{T}\ R'$. By Lemma 58 we can alpha-rename this, giving $(y, (y{=}x \mid ux.P^*)\{x/y\}\{y/x\})\ \mathcal{T}{=}\ R'$. Then it is straightforward to apply the restriction x, followed by the restriction y. The second scope law is more straightforward.

3. Reflection and transitivity are straightforward. So are the laws for fusion equivalence and interchange.

4. The only interesting case for congruence is when, given $P \equiv Q$, we deduce $(x)P \equiv (x)Q$. Now, suppose $(\tilde{w}, (x)P)\ \mathcal{T}\ P'$. By definition of $\mathcal{T}$, $x \notin \tilde{w}$. The restriction might be playing one of three roles. Let us consider the first role, of binding some input command. Then $(\tilde{w}x, P)\ \mathcal{T}\ P'$. From the induction hypothesis, $(\tilde{w}x, Q)\ \mathcal{T}\ Q'$ such that $Q' \equiv P'$. Therefore $(\tilde{w}, (x)Q)\ \mathcal{T}\ Q'$ as desired. The other roles are similar. □

With these two lemmas we can establish that the relation $\mathcal{T}$ can indeed be used to characterise piability:

Corollary 62 *P_ϕ is piable if and only if $\exists Q_\pi : (\emptyset, P_\phi)\ \mathcal{T}\ Q_\pi$.*

Proof. In the forward direction, the definition of piability says that there exists a Q_π such that $P_\phi \equiv Q_\pi^*$. Now we know from Lemma 60 that there exists a Q'_π such that $(0, Q_\pi^*)\ \mathcal{T}\ Q'_\pi$. Finally, since $P_\phi \equiv Q_\pi^*$, and from Lemma 61, there also exists some Q''_π satisfying the lemma.

In the reverse direction, suppose $\exists Q_\pi : (\emptyset, P_\phi)\ \mathcal{T}\ Q_\pi$. By Lemma 59, $Q_\pi^* \equiv P_\phi$, so P_ϕ is indeed piable. □

Corollary 63 *If P_ϕ is piable, and $P_\phi \equiv P'_\phi$, then P'_ϕ is also piable.*

We will now see that the translation $(\_)^*$ preserves barbs and reactions.

Lemma 64

1. *$Q_\pi^* \xrightarrow{\mu}$ implies $Q_\pi \xrightarrow{\mu}$.*
2. *$Q_\pi^* \xrightarrow{\tau} P'_\phi$ implies there exists a Q'_π such that $Q_\pi \xrightarrow{\tau} Q'_\pi$ and $P'_\phi \equiv Q'^*_\pi$,*

Proof. The first part is a straightforward induction on the structure of Q_π.

For the second part, we perform an induction on the derivation of $P \xrightarrow{\tau} P'$ with the induction hypothesis that

- if $(\tilde{x}, P)\ \mathcal{T}\ P_1$ and $P \xrightarrow{\tau} P'$ then there exists a P_1' such that $(\emptyset, (\tilde{x})P')\ \mathcal{T}\ P_1'$.

Note that the assumption of this hypothesis uses $(\tilde{x}, P)$ while the consequent uses $(\emptyset, (\tilde{x})P')$. The names $\tilde{x}$ have changed place: in the assumption, they are all being used to bind input; in the consequent, some will discharge the resulting fusions while others will just be restrictions.

There are four cases to consider: the base case, congruence with restriction, congruence with parallel composition, and structural congruence. We show the base case, and parallel composition; the others are straightforward.

1. Say $(\tilde{z}, \overline{u}\tilde{x}.P \mid u\tilde{y}.Q)\ \mathcal{T}\ R$. Therefore $\tilde{z}$ must be just $\tilde{y}$. We must demonstrate an R' such that $(\emptyset, (\tilde{y})(\tilde{x}{=}\tilde{y} \mid P \mid Q))\ \mathcal{T}\ R'$. This is straightforward, since the restrictions just discharge the fusions.

2. Say $(\tilde{z}, P|Q)\ \mathcal{T}\ R$ and $P|Q \xrightarrow{\tau} P'|Q$. Now $\tilde{z}$ can be partitioned into $\tilde{z}_1$ and $\tilde{z}_2$, with $(\tilde{z}_1, P)\ \mathcal{T}\ P_1$ for some P_1. Note that every name in $\tilde{z}$ is fulfilling the role of binding input. Therefore, the partitioning will not affect the ability of P and Q to interact. By the induction hypothesis, $(\emptyset, (\tilde{z}_1)P')\ \mathcal{T}\ P_1'$ for some P_1'. The desired result can be constructed easily.

□

Lemma 65 *For terms Q_π and Q'_π in the pi calculus,*

1. $Q_\pi \xrightarrow{\mu}$ *implies* $Q^*_\pi \xrightarrow{\mu}$.
2. $Q_\pi \xrightarrow{\tau} Q'_\pi$ *implies* $Q^*_\pi \xrightarrow{\tau} Q'^*_\pi$.

Proof. Straightforward, since the explicit fusion calculus is a generalisation of the pi calculus. □

Corollary 66 (Bisimulation) $Q_\pi \dot{\sim}_b Q'_\pi$ *if and only if* $Q^*_\pi \dot{\sim}_b Q'^*_\pi$

Weak bisimulation is also preserved.

Congruence. We now turn to congruence. We show that if two pi terms are barbed bisimilar in all pi contexts (i.e. shallow barbed congruent), then their translations are barbed bisimilar in all piable explicit fusion contexts. As explained in Section 4.1, this result is enough for practical purposes. We also show that general explicit fusion contexts are strictly more discriminating than piable explicit fusion contexts. Let us first consider some features of piable contexts.

Fusing power. No pi context can ever make x and y equal in the program $Q_\pi = (xy)(\overline{u}xy \mid \overline{x} \mid y)$, and so direct reaction between $\overline{x}$ and y is impossible. However, the explicit fusion context $E_\phi = uzz$ will make them equal, so allowing a direct reaction. This difference is the reason why explicit fusion contexts are more discriminating than pi contexts.

I conjecture that, for weak bisimulation, pi contexts are *not* less discriminating that explicit fusions. That is because weak bisimulation does not distinguish whether x and y are directly equal, or merely connected by an equator. I leave this as an open problem, because, as explained in Section 3.9, weak bisimulation is difficult.

Non-universality. The pi context $u(u).\_$ does not have a corresponding context E_ϕ in the explicit fusion calculus, such that $\forall Q_\pi : E_\pi[Q]^* \equiv E_\phi[Q^*]$. This is because, to translate the pi context, we need to pick a fresh name u', giving $(u')u.(\_\{u'/u\})$. And it is not possible to choose a u' which will be fresh for every Q_π which might be placed in the hole. Instead, this pi context really corresponds to a family of explicit fusion contexts, each one hypothecated to a set of permissible names in the hole.

I conjecture that, nevertheless, piable contexts are exactly as discriminating as pi contexts. That is to say, the embedding of the pi calculus into the explicit fusion calculus is complete as well as sound. To prove this would require emulating the substitutive context $\_\{u'/u\}$, as in the following.

Non-substitutability. Consider $E_\phi = (x)(x{=}y \mid \_)$. It is true that for all P_ϕ that are piable, $E_\phi[P_\phi]$ is also piable. However, the context E_ϕ itself is not piable. That is because it amounts to a substitutive context $\_\{y/x\}$, and the pi calculus does not have directly substitutive contexts. Instead, in the pi calculus, such substitutions must be emulated: $E_\pi = (u)(\overline{u}y \mid u(x).\_)$.

Given this issue of non-substitutability, one might ask why we defined piable contexts as we did:

- *(shallow)* a context E_ϕ is piable when there exists an E_π such that for all Q_π, $E_\pi[Q_\pi]^* \equiv E_\phi[Q_\pi^*]$,

rather than

- *(reduction-closed)* a context E_ϕ is reduction-closed piable when for all P_ϕ that are piable, $E_\phi[P_\phi]$ is also piable.

The names given to the two definitions are suggestive of the reason, which is as follows. We are working with shallow congruence. This places the programs in an initial context: this models the practical situation where we are using the program as a library routine within some larger program (it's context). Therefore this context will not have reacted yet, and is still just a direct translation of some pi program: i.e. it is a piable context.

However, if we had used reduction-closed congruence, we would be modelling the situation where partially-executed programs can be placed in partially-executed contexts. Such partially-executed contexts might perhaps not be piable (as per the example of non-substitutability) even though the system as a whole might still be piable (as per the reduction-closed definition). This is why reduction-closed congruence would use its different version of piability.

Proposition 67 (Shallow soundness) *For all terms Q_π and Q'_π in the pi calculus, $\forall E_\pi : E_\pi[Q_\pi] \dot{\sim}_b E_\pi[Q'_\pi]$ implies $\forall E_\phi piable : E_\phi[Q_\pi^*] \dot{\sim}_b E_\phi[Q_\pi'^*]$.*

Proof. We are given that

$$\forall E_\pi : E_\pi[Q] \dot{\sim}_b E_\pi[Q'].$$

Also, by definition of piability, we know that for all piable E_ϕ,

$$\exists E_\pi : E_\pi[Q]^* \equiv E_\phi[Q^*] \wedge E_\pi[Q']^* \equiv E_\phi[Q'^*].$$

From these two, and Corollary 66, we get that for all piable E_ϕ,

$$\exists E_\pi : E_\phi[Q^*] \equiv E_\pi[Q]^* \dot{\sim}_b E_\pi[Q']^* \equiv E_\phi[Q'^*]. \qquad \square$$

Example 68 *There exist pi calculus terms Q_π and Q'_π which are barbed bisimilar in all pi contexts, but there is an explicit fusion context E_ϕ such that $E_\phi[Q^*_\pi]$ and $E_\phi[Q'^*_\pi]$ are not barbed bisimilar.*

Proof. We will use the fact that, in the pi calculus, the restriction $(xy)(\overline{u}xy \mid P)$ ensures that the names x and y will never become fused in P. By contrast, in the explicit fusion calculus, the context $\_ \mid uzz$ will make them fused. We need to find two terms P and Q which have the same behaviour when x and y are different, but different behaviours when they are the same. Boreale and Sangiorgi have provided two such terms [8].

In the following, we will write $\tau.P$ as an abbreviation for $(u)(\overline{u} \mid u.P)$, and use E as an abbreviation for $(xy)(\overline{u}x \mid \overline{u}y \mid \_)$.

$$\begin{aligned} P &\stackrel{\text{def}}{=} !\overline{y}.x.\tau.z \mid !x.\overline{y}.\tau.z \\ Q &\stackrel{\text{def}}{=} !(w)(\overline{y}.\overline{w} \mid x.w.z) \end{aligned}$$

Note that when $x = y$, then Q has a barb on z after two steps, but P has one only after three. Therefore $E[P]$ and $E[Q]$ are not barbed congruent in the explicit fusion calculus. However, they are in the pi calculus. □

Conclusions. This concludes our proofs of the embeddings of the fusion calculus and the pi calculus into the explicit fusion calculus. Let us review what has been accomplished in this chapter.

We have proved that the embedding of the fusion calculus into the explicit fusion calculus is both sound and complete—i.e. it is fully abstract. This means that we can use proof techniques from the explicit fusion calculus, such as efficient bisimulation, to prove properties about programs in the fusion calculus. However, the explicit fusion calculus allows some terms to react that cannot react in the fusion calculus. This means that the explicit fusion calculus is not well suited as an implementation of the fusion calculus. Indeed, an implementation of the fusion calculus is generally awkward, since its reaction is not local.

We have however proved that reaction and barbs in the pi calculus correspond exactly to reaction and barbs of translated terms in the explicit fusion calculus. We have further proved that the embedding of the pi calculus into the explicit fusion calculus is sound with respect to translated contexts. In Chapter 6 we will extend this result to the fusion machine (which is based on the explicit fusion calculus), to prove that the fusion machine is a sound implementation of the pi calculus.

Chapter 5

The fusion machine

The *fusion machine* is a distributed, concurrent implementation of the explicit fusion calculus and the pi calculus. This chapter explains it through diagrams and examples. The working programmer will find these sufficient to implement the machine. Proofs of correctness, and a formal algebraic notation, are found in Chapter 6 (page 94).

This chapter has six sections. Each section describes a progressively more powerful fusion machine:

5.1. *Operation.* A basic machine, able to implement input and output commands in parallel.

5.2. *Restriction.* How restriction in the calculus can be modelled in the machine by a registry of free names.

5.3. *Deployment.* Support for the prefix operator, through the deployment of continuations.

5.4. *Replication.* Support for the replication operator. The machine can now execute all programs in the explicit fusion calculus.

5.5. *Co-location.* Optimisations that become possible when two channel managers happen to reside on the same physical computer.

5.6. *Fairness and failure.* Discussion of further work.

I invented the fusion machine in an attempt to write a native compiler for the pi calculus: that is, a way to compile each part of a pi program into machine code that executes directly, rather than into data that is manipulated by a runtime interpreter. My intuition was that channel management in the pi calculus seems similar to memory management in a conventional programming language: therefore the same techniques developed for optimising compilers should also apply to an implementation of the pi calculus. Now in a conventional language it is more efficient for data to be stored on the stack than in the heap. This is because a computer can access the stack directly, whereas heap access involves an indirection. (Direct stack access amounts to using De-Bruijn indices; heap access amounts to looking up a name.) Also, using a stack avoids the need for garbage collection. But we have a problem applying this principle to the pi

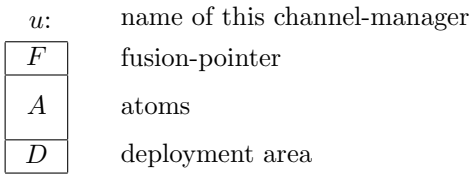
calculus: the stack cannot be used for variables with dynamic scope, and yet dynamic scope is a fundamental part of the pi calculus. My solution is to place code inside the channel managers. This allows the code to refer directly to its own channel, without the need for a name lookup.

The fusion machine therefore uses channel managers. One might ask whether it is possible to make a distributed implementation of the pi calculus without channel managers. Schwartz [61] has shown how to do without channel-managers in the special case where the connections between programs never change. However, his technique takes a large number of handshakes, and in any case it cannot be extended to the full pi calculus.

5.1 Operation

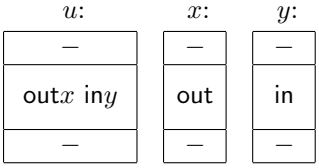
The fusion machine is a collection of located channel-managers which run in parallel and which interact. Each channel-manager contains fragments of a program. Some fragments it sends across the network to other channel-managers; other fragments it executes locally. Through these two operations it implements the explicit fusion calculus and the pi calculus. In these calculi, the only work done by local execution is to fuse names. However, it is easy to integrate other languages into the fusion machine, to do other local work.

Assume a set of channel names ranged over by u, v, w, x, y, z with a total order. The order might use 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. (Later in the chapter, we will introduce co-located channels.) Each channel-manager is made from three parts: a fusion-pointer (F), atoms (A) and a deployment area (D). It is pictured as follows:



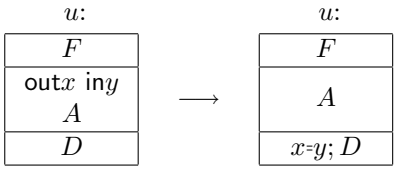
The *fusion pointer* is either the name of some other channel, or empty. If it does point to some other channel, then the atoms contained in the channel-manager may be sent across the network to that other channel. The *atoms* are a collection of output atoms $\mathsf{out}x$ and a separate collection of input atoms $\mathsf{in}x$. In general they may be polyadic (communicating several names), although this chapter uses only monadic (single names) for simplicity. The *deployment area* is a collection of fusions $x{=}y$. From each collection, we assume the implementation is able to pick an arbitrary element. Let m range over $\{\mathsf{out}, \mathsf{in}\}$.

As an example, the following diagram represents the term $\overline{u}x \mid uy \mid \overline{x} \mid y$:

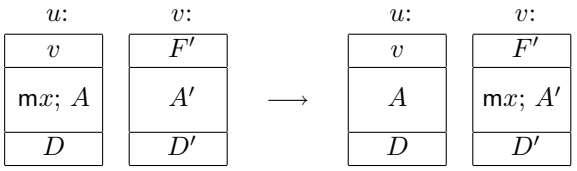


(17)

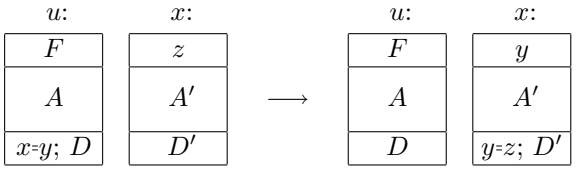
In the basic machine without prefixes, there are just three transition rules. The explanation for why the rules are as they are, and how just three rules are sufficient, is subtle. We give the rules first, then illustrate how they execute the above program, and then explain them.



(interact)



(migrate)

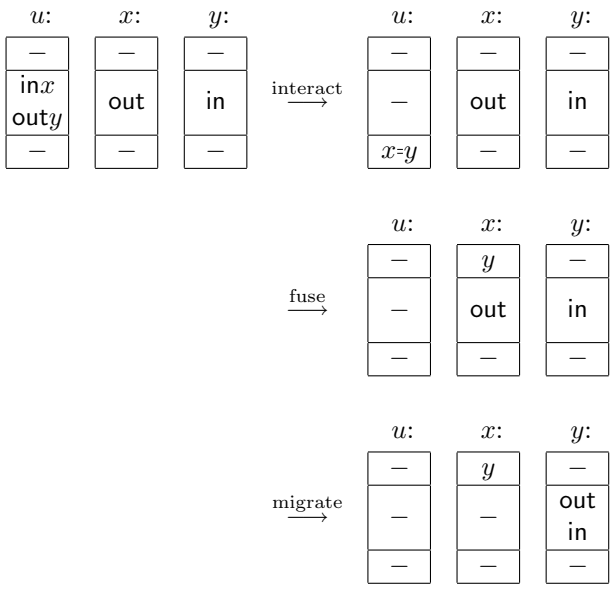


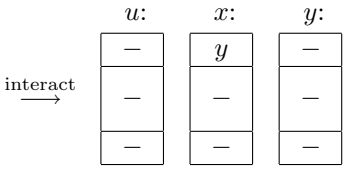
(fuse)

In the final rule we assume $x < y$ in the total order on names. This means that lesser names always point to greater names, and (as explained below) will lead to the fusion pointers forming a tree. If in the final rule x started with an empty fusion pointer rather than z, then we omit $y{=}z$ in the result. A fusion $x{=}y$ in the deployment area is equivalent to $y{=}x$.

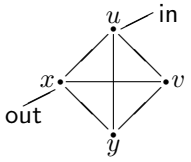
To illustrate the execution of the machine, we now show the execution trace of Example 17

(page 78). The following steps represent $\overline{u}x \mid uy \mid \overline{x} \mid y \longrightarrow x{=}y \mid \overline{x} \mid y \equiv x{=}y \mid \overline{y} \mid y \longrightarrow x{=}y$, assuming $x < y$.

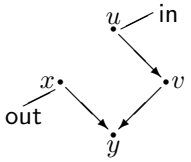




We now explain the role of the fusion-pointer inside each channel-manager, and explain the rules. The fusion machine operates around fusion-pointers: interaction creates them; fusion deploys them into fusion-pointers; and migration uses them. To understand the role of a fusion pointer, first consider the term $u{=}v{=}x{=}y \mid \overline{x} \mid u$ in the explicit fusion calculus. The fusions generate an equivalence relation on names. In this example they allow reaction, since u is interchangeable with x. We might picture the term as follows:



However, in a distributed setting, u and $\overline{x}$ are different names, and hence at different locations. In order that the atoms at u and x can react together, we must send them to a common location on the network. The decision as to where to send them must be taken locally: the channel-manager at u must choose where to send its input atom, and the channel-manager at x must choose where to send its output atom. The problem is to find an algorithm and data structure that allows such local decisions. The solution used in the fusion machine is to represent the equivalence relation by a rooted tree. Then each channel in the tree can send its atoms to its parent, and the atoms are guaranteed to arrive, eventually, at the same location. In the following picture, both atoms will migrate to y where they can react:

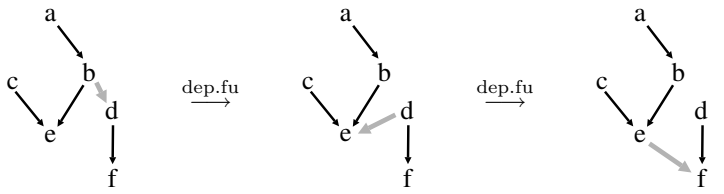


For each channel, the pointer to its parent in the tree is stored as its fusion-pointer F. (In the diagrams above, children are at the top and parents at the bottom.)

The fusion transition should be understood in the following sense. Interaction gives rise to a fusion of two names, and this fusion must ultimately merge their corresponding trees. The algorithm for merging is complicated. So, we break it into smaller steps. The algorithm's intermediate state between steps is stored entirely as fusions in the deployment area. And the algorithm's invariant is that the tree respects the order on names, such that greater names are placed closer to the root. Therefore, each fusion transition takes a fusion progressively closer to the root, and the algorithm necessarily terminates.

A sequence of fusion transitions is illustrated below. In these pictures, a thick grey line from b to d indicates that some channel-manager contains the fusion $b = d$ in its deployment area. We assume the standard alphabetical order

on names. Observe how the fusion is moved progressively closer to the root (at the bottom) of the tree, until finally it can be satisfied.



This algorithm for merging trees is similar to the Union/Find algorithm of Tarjan [67]: it also represents an equivalence relation as a set of trees, with each tree corresponding to an equivalence class, and it forms the union of two equivalence classes by making one tree a child of another.

There is no handshaking involved in any of the transitions: that is to say, each transition can be accomplished with just a single message from one channel to another, and no acknowledgement or reply is needed. In all three rules listed on page 79, the channel-manager at u makes a local decision to do something and then perhaps sends a message. For migration, the message is 'please accept this migrant'. For fusion, the message is 'please fuse yourself to y'. (Although they never shake hands, fusion machines are invariably polite!) It would be strictly more correct, although no clearer, to write the sending of a message as one transition, and the reception as another. Then each rule could be written involving just a single machine and the *ether*—this ether representing the network fabric through which messages propagate. We might also account for message failure by deleting random messages from the ether. However, I will only address questions of efficiency and not of failure.

5.2 A registry of free names

We now consider restriction in the pi calculus and explicit fusion calculus, and how to implement it. The role of restriction in the calculi is easy to understand: it is a binder; it allows for alpha-renaming; through scope-extrusion, it indicates how far the knowledge of a name might have escaped; it allows two terms to be placed in parallel without name-clashes; and it can make a channel local, so that no external program can rendezvous at it.

In an implementation, the important role concerns the separate compilation of programs. Suppose that one program is already running, with its various channels at their IP and port numbers. And suppose that we later wish to write a second program, and have it interact with the first. To do this, the second program must know which IP and port numbers are used by the first. There are two possible techniques to discover these numbers. First, there might be some registrar of published channels. The Internet's Domain Name Service is such a registrar: after noticing the name 'www.wischik.com' in a dissertation, we can ask the Domain Name Service for the IP number for that name. (As for the port number, Internet convention is that web services are provided on port 80). To find names in this way obviously requires the original program to have published its details in the registrar. The second way to discover IP and port numbers, is simply to scan through all possible numbers. This technique is typically used to find vulnerable channels and gain illegal access to machines.

We treat free names, then, as a model for this central registry. All free names are in the registry. All names bound by a restriction are not in the registry. We will write $(\!|x|\!)$ to indicate a channel-manager x which is not in the registry: therefore, no additional programs can rendezvous at x, unless they are told its name through some other communication.

The other roles of restriction are not so relevant. The fusion machine will create fresh names that are globally unique. This is easy to do in practice, but awkward in theory, and the calculus roles of alpha-renaming and binding are just a model for it. The calculus role of avoiding name-clashes is not so relevant for an implementation: IP numbers are normally unique, by construction, so plugging two networks together does not result in clashes. The final role for restriction, of preventing an external program from even guessing an IP and port number, is only relevant in a security-conscious application designed to be robust even in the presence of port-scans. In any case, the explicit fusion calculus has no command for guessing IP and port numbers, so the issue does not arise when implementing the calculus.

Consider the final state of the execution trace on page 79:

u:	x:	y:
–	y	–
–	–	–
–	–	–

(18)

If it had happened that the names u and x were private, not in the registry, we would instead have had this:

$(\!\|u\|\!)$:	$(\!\|x\|\!)$:	y:
–	y	–
–	–	–
–	–	–

(19)

Here, no additional program can ever rendezvous at channels u or x again: we might therefore delete them both. A simple reference-counter would suffice to tell whether an unregistered channel-manager can be deleted. I suspect that a *delete* command would also be useful in practice.

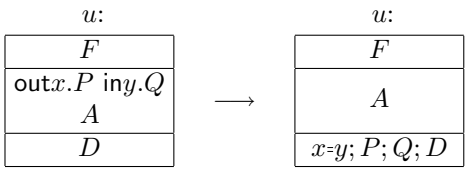
Remember that fusions respect the total order on names: they always go from the lesser name to the greater. If in the above example y had been a lesser name than x, then x would always be referenced and so could never be freed. This is wasteful. To avoid it, the registered names should rank higher in the total order than unregistered names.

It must be stressed that the registry of names has *no bearing on the operation of the machine*. The machine operates in exactly the same way regardless of which names are in the registry. Although we will write all the operations of the machine with registered names, they apply equally to private names. It would in fact be possible to define the machine without any registry at all, as discussed in the following chapter. However, we shall retain the registry, as a point of familiarity with the calculus and also as a technical aid in defining barbed bisimulation.

5.3 Deployment

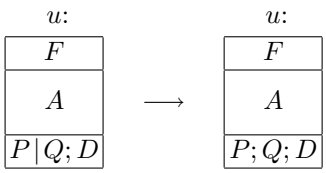
We now add rules to the machine so as to support the deployment of arbitrary terms. There are two cases when deployment is used: first, when pre-deploying a program which has been fragmented for efficiency; second, when deploying some guarded term that has been newly liberated after a reaction. In fact, both cases use exactly the same deployment rules. The additions to the machine are straightforward: we allow prefixed input and output atoms; and, rather than just explicit fusions in the deployment area, we allow arbitrary terms from the calculus. We will focus on explicit fusion calculus, but will also show how to deploy terms from the pi calculus.

Let the atoms (A) now be a collection of output atoms $\mathsf{out}x.P$ and a collection of input atoms $\mathsf{in}x.P$, both prefixing some term P in the explicit fusion calculus. Let the deployment area (D) now be a collection of arbitrary terms in the explicit fusion calculus, rather than just explicit fusions. We modify the interaction transition to account for the prefix, and add one deployment transition for each construct in the calculus. The deployment of explicit fusions is as before; we give it the uniform name *(dep.fu)* rather than its previous name *(fuse)*. The new transitions are as follows.

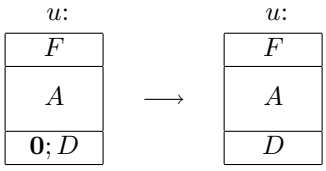


(interact)

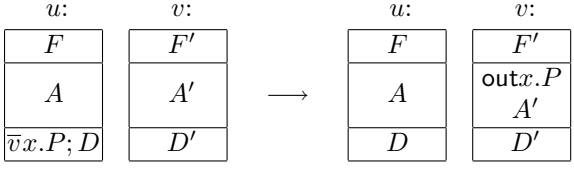
This interaction transition is like the previous interaction transition, but augmented to operate upon atoms which guard a term. After reaction, the term is placed in the deployment area, where it can be deployed further.



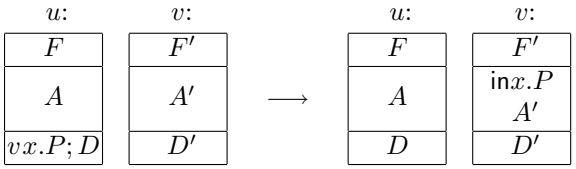
(dep.par)



(dep.nil)

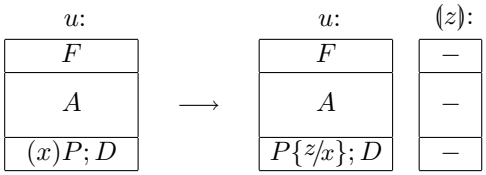


(dep.out)



(dep.in)

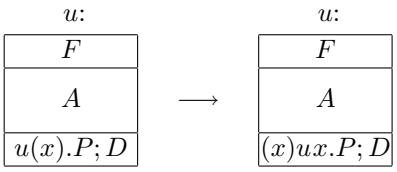
These deployment transitions are all straightforward. They take a program fragment from the deployment area, and either break it down further or send it to the correct place in the network. To send it, we can use exactly the same migration messages as before—although each message now includes a continuation, and may well be large. Turner uses similar deployment transitions in his uniprocessor abstract machine (described in Section 1.3, page 10 of this dissertation).



(dep.new)

This transition deploys a restriction. It does this by creating a new, unique channel-manager. Here, we represent the fact that the channel-manager is new and unique by calling it with a *fresh* name—one that is globally unique.

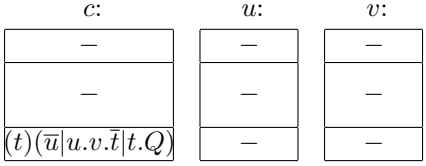
We might also add a further deployment transition, as below. It is not necessary for the expressiveness of the fusion machine, but it conveniently allows the machine to execute pi calculus terms directly. In effect, it translates terms from the pi calculus into the explicit fusion calculus.

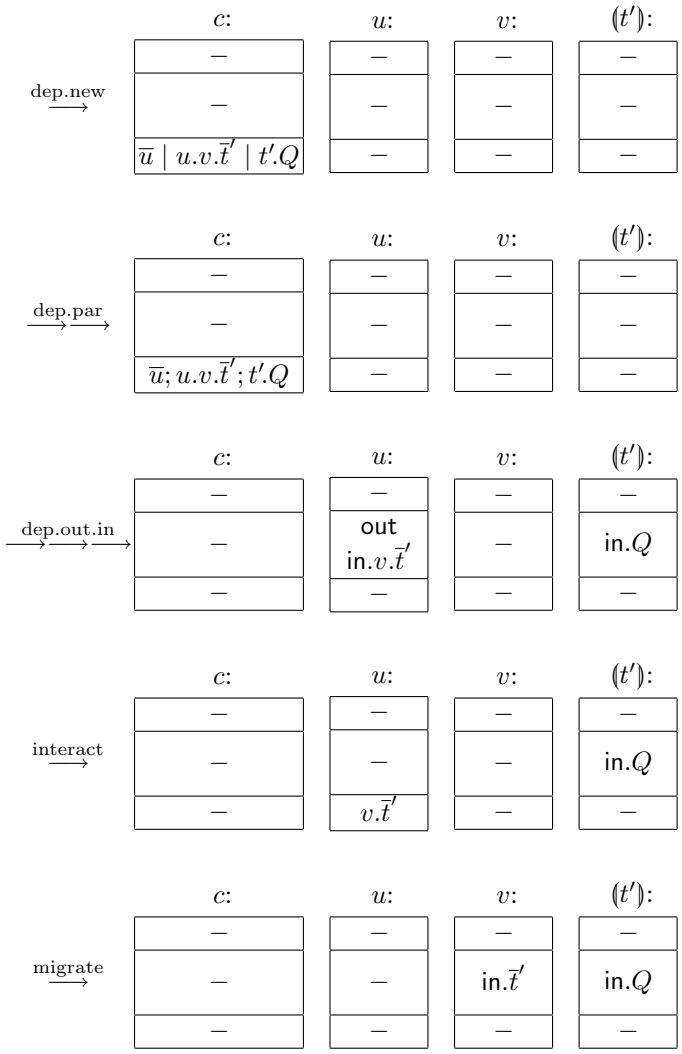


(dep.bound)

Now that machines include deployment, we can represent a programmer who writes a program P and runs it on the fusion machine. Perhaps this programmer loads his program onto the network in Cambridge, location c. The initial state is simply $c;[-,-,P]$. If the optimising compiler had seen fit to fragment the program P into two parts $(t)(P_1 \mid P_2)$, so as to reduce total message size, then the initial state is $c{:}[-,-,(t)(P_1 \mid P_2)]$. Thus, exactly the same technique can be used to pre-deploy program fragments, as to deploy terms newly liberated by reaction.

As an example, let $P = \overline{u} \mid u.v.Q$, and suppose it has been fragmented into $(t)(\overline{u} \mid u.v.\overline{t} \mid t.Q)$ where t is not free in $u.v.Q$. Let t' be fresh.





One thing to observe in this example is how the fragment $t.Q$ only migrates once, to its final destination t. If we had run the program without fragmenting it, then Q would have had to migrate twice—first to u, and then to v. If Q happens to be a large program, slow to send over the network, then our fragmentation technique has gained efficiency.

Note that the exact order of deployment depends on the (arbitrary) order in which terms are chosen from the deployment area. The execution trace above is an arbitrary trace—not the only one. However, the above example is confluent no matter which trace is chosen. In the following chapter we will see that all deployment transitions are confluent: it is only through interaction that non-determinism arises, just as in the calculus.

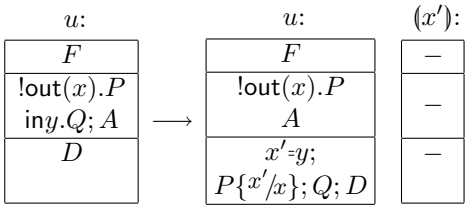
5.4 Replication

We now add rules to the machine so as to support replication. The idea is that replicated atoms can interact in exactly the same way as normal atoms, except

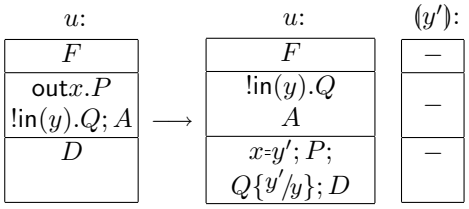
that they are not used up by the interaction. Now that the machine includes replication, it can implement the full explicit fusion calculus and pi calculus.

The machine incorporates several simplifying assumptions, notably in its use of *guarded replication* (i.e. the replication of an input or output command). We present the machine with replication first, and discuss the assumptions afterwards.

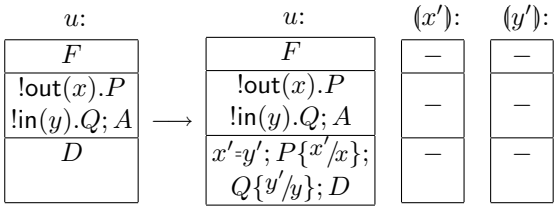
Let there be two additional kinds of atoms: replicated output $!\mathsf{out}(x).P$, and replicated input $!\mathsf{in}(x).P$. Again we write monadic restriction for simplicity; polyadic restriction is straightforward. In these atoms, x is bound in P. We have three new interaction transitions, according to whether the output or input or both are replicated:



x' fresh (int.rep.out)



y' fresh (int.rep.in)



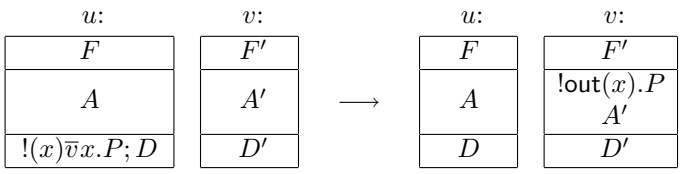
x',y' fresh (int.rep.both)

Although the transitions look complicated, they can be understood quite simply. Each one is an interaction in the style of *(interact)*. But if an atom is replicated, then it is not consumed by the interaction, and is instead kept. If any names are restricted in the replication, then they are created fresh in the same way as *(dep.new)*.

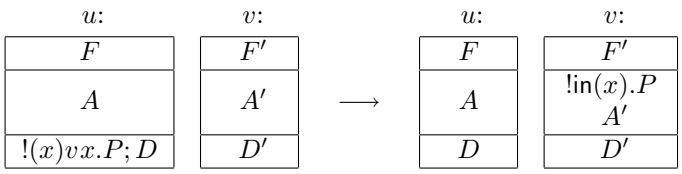
It is unfortunate to have to combine so many features—interaction, duplication, restriction—into a single command such as $!\overline{u}(x).P$. Indeed, the pi calculus and the explicit fusion calculus manage to treat each feature separately. For example, in $!(x)\overline{u}x.P \equiv (x)\overline{u}x.P \mid !(x)\overline{u}x.P$, the structural congruence performs the duplication; and restriction and interaction can then be handled separately in the normal way. But it is not possible to implement this structural congruence rule as a transition $u{:}[-,-,!P] \longrightarrow u{:}[-,-,P;!P]$, since this transition would allow for an unlimited number of copies to be made.

It is standard instead to use only guarded replication: i.e. replication of the form $!(\tilde{x})\mu\tilde{y}.P$. The copies can then be created only on demand. It is because

of this that we must combine the features into a single transition rule. In fact, we have used an even simpler form, where the same names are restricted ($\tilde{x}$) as communicated ($\tilde{y}$). This simplicity does not come at the expense of generality. For consider the general form $!(xz)\mu zy.P$, in which some names are restricted that are not communicated, and some are communicated that are not restricted. This is equivalent to $!(z'y)\mu z'y.(x)(z{=}z' \mid P)$. Other authors simplify still further by disallowing replicated output



(dep.rep.out)



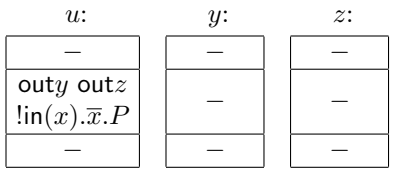
(dep.rep.in)

These two transitions are for the deployment of replicated input and output commands. One might also add a transition for the deployment of replicated bound input, so that the machine can execute a pi calculus term directly, but we omit it here.

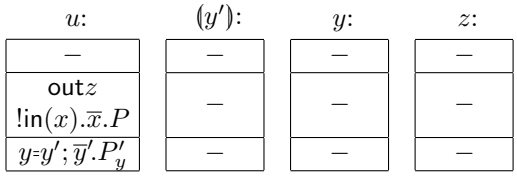
As an example of replication, we illustrate the following execution.

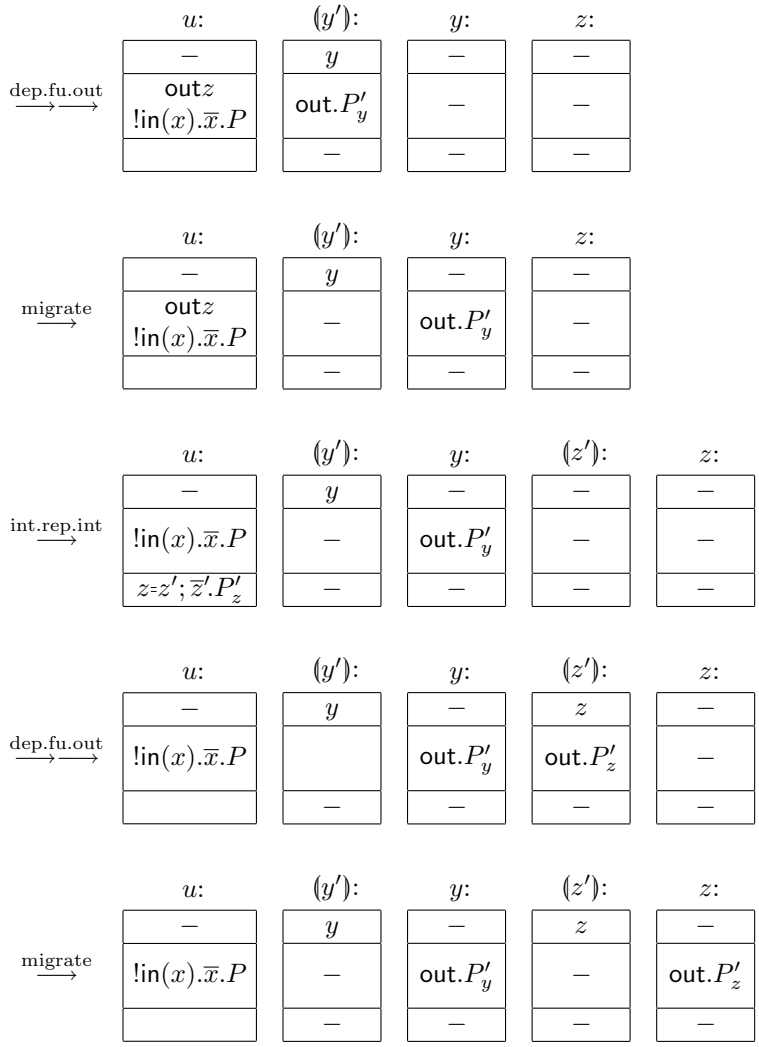
$$\begin{aligned} \overline{u}y \mid \overline{u}z \mid !(x)ux.\overline{x}.P &\longrightarrow \overline{u}z \mid !(x)ux.\overline{x}.P \mid \overline{y}.P\{y/x\} \\ &\longrightarrow !(x)ux.\overline{x}.P \mid \overline{y}.P\{y/x\} \mid \overline{z}.P\{z/x\} \end{aligned}$$

In writing this example as a fusion machine, let y' and z' be fresh names, with $y' < y$ and $z' < z$. The machine is a little tedious to write out in full, so we use the abbreviations $P'_y = P\{y'/x\}$ and $P'_z = P\{z'/x\}$.



int.rep.in $\longrightarrow$





Note that the channels y' and z' were created fresh. We therefore know that no one else has a reference to them, and so they can be garbage-collected.

One particular optimisation is apparent in this example. Rather than creating a local channel y' and then migrating things from y' to y, we could instead just send them directly to y. This amounts to an immediate substitution $\overline{u}y \mid !u(x).P \longrightarrow P\{y/x\}$ in the style of the pi calculus, rather than delayed substitution in the style of the explicit fusion calculus. However, note that we cannot wholly dispense with delayed substitution: it is still needed for fragmentation, as explained in Section 1.4 (page 12). In effect, while the pi calculus is a special (and probably common) case, the explicit fusion calculus is the general case.

5.5 Co-location

Suppose that two channel-managers happen to reside at the same location—in particular, that they share an address space. Perhaps they are both running on

a single user's desktop computer. Two optimisations are possible, to do with migration and threads of execution.

Let us draw co-located channel-managers as physically adjacent:

u:	v:
F	F'
A	A'
D	D'

The first optimisation concerns the case that u is fused to v (or vice versa): we can migrate all atoms from u to v in constant time. To achieve this, let both collections of atoms—output and input—be stored as linked lists with tail pointers. Then to splice u's list onto the end of v's list is easy: just make the previous tail of v point to the head of u, and make the new tail of v point to the tail of u. The transition for constant-time migration is as follows:

u:	v:
v	F'
A	A'
D	D'

$\longrightarrow$

u:	v:
v	F'
$-$	$A; A'$
D	D'

(migrate.at)

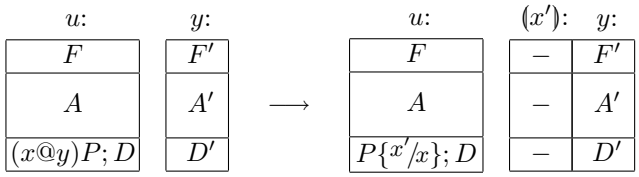
We use this form of constant-time migration in the following chapter to prove that the encoding of fragmentation in the explicit fusion calculus does not cost extra inter-location messages.

The second optimisation concerns threads of execution. If there are several channels at the same physical location, then we no longer need a separate thread of execution for each. A single thread might handle all the channels in round-robin fashion. Indeed, we can even spawn new threads to handle some of the channels, or retire threads, at will. This might be particularly useful in a mixed-language environment: In a program such as $\overline{u}x.C$, perhaps the code C is very slow and should be run in a new thread so that other interactions can continue; or perhaps it is a blocking operating system call and so requires its own thread.

Consider the degenerate case of a non-distributed machine which has all its channels in the same physical location. If we use just a single thread of execution, in round-robin fashion, then the machine becomes substantially the same as the uniprocessor implementation (Section 1.3, page 10). The advantage of the fusion machine is that it can scale easily to multiple threads on a single system, and distributed systems; the uniprocessor machine is not scalable.

We now address the question of how co-location might be programmed, and next how it should be implemented. Presumably, the programmer would wish to decide the location of some channel. The obvious time to do this is at the time the channel is created: i.e. the restriction command. To this end we augment the explicit fusion calculus with a new kind of restriction, *located restriction*, written $(x@y)P$. This is to indicate that the new channel x should be created physically adjacent to y. The deployment transition for located restriction is as

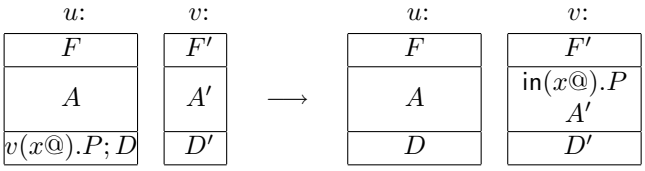
follows.



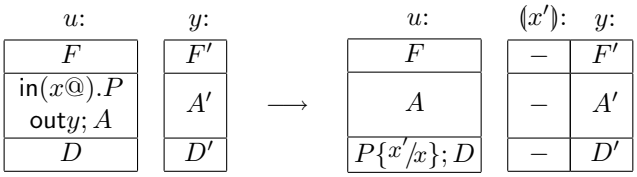
(dep.new.at)

This location command is the only one treated formally in the following section. However, it is inadequate to express one useful programming idiom. Say we have a program $(x)ux.P \mid \overline{u}y \mid \overline{u}z$, and we wish the fresh name x to be located next to either y or z depending on which reaction happened. That is, if the reaction involved $\overline{u}y$ then the result will be $(x@y)(x{=}y \mid P) \mid \overline{u}z$; and if the reaction involved $\overline{u}z$ then the result will be $(x@z)(x{=}z \mid P) \mid \overline{u}y$. The decision as to the location of x *cannot be made until the the time of the interaction.* For an example application which needs this sort of delayed decision, consider a mobile agent. Say the agent receives a request, and then migrates to the originator of that request to deliver the answer in person. But it cannot know where to migrate, until after it has received the request.

Located restriction as described above only allows a decision to be made before interaction. Therefore it is not suitable for the mobile agent. Instead, we need a special form of command where interaction and located-name-creation happen at the same moment. So, we also add *located bound input* $u(x@).P$ to the calculus. This means that, upon reaction with some $\overline{u}y.Q$, the new channel x should be created at the same location as y. The corresponding atom in the machine is $\mathsf{in}(x@).P$. The two additional transitions are as follows—one to deploy the bound input command, and one to interact with it.



(dep.bound.at)



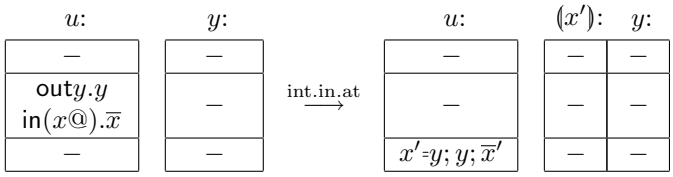
(interact.at)

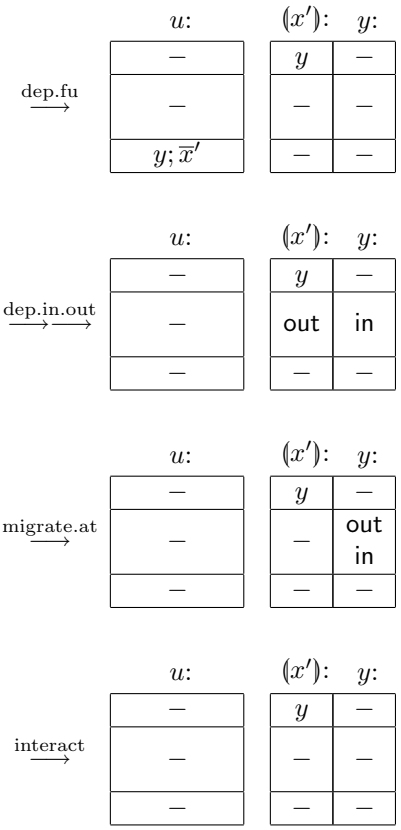
The symmetric extension to located bound output is obvious, so we omit it.

We illustrate the execution of located bound input with the example

$$u(x@).\overline{x} \mid \overline{u}y.y \longrightarrow (x@y)(x{=}y \mid \overline{x} \mid y) \longrightarrow (x@y)(\mathbf{0}).$$

Let x' be fresh, with $x' < y$.





Observe that, in this example, no atoms migrate to the channel x' until after it has been created. This might seem trivial, but in fact it is not. Let us spell out the chain of dependencies: before u can send anything from its deployment area, it must first perform the substitution $\{x'/x\}$; and it cannot perform the substitution until it knows the name x'; and it cannot know the channel-name x' until that channel has been created. The question is *who* should create x'? More precisely, who should generate the globally unique identifier x'? And who should allocate storage space and create a thread of execution for it?

The obvious answer is that, since x' will be at the same location as y, it should be created by y. But this is undesirable because it would involve handshaking: u would have to send a message to y asking it to create a new channel, and then y would create the channel and send back a message to announce the channel's name.

However, we can avoid handshaking if we suppose that u is able *locally* to create fresh unique identifier x', even though x' will be co-located with y. This is possible if each name is tagged with the location that created it (in this case, u) and a time-stamp. Then, each migrating atom can be sent from u to y with a short note: 'Please actually place me in x', not y; and if x' has not yet been created, then create it'. I call this *lazy channel creation*. The ability to locally create identifiers that are globally unique is a useful one in practice. There is an international standard for globally unique identifiers, detailed in [27, 36]. This standard was actually invented to support various different aspects of *remote procedure calls*—which are a form of synchronous rendezvous.

5.6 Fairness and failure

Fairness means that things ready to interact do not get starved of interaction. We define *atomic fairness* as follows: *if it remains perpetually possible for an atom to interact, then eventually it will.* That is, if some atom always has some complementary atom in the same equivalence class, then eventually it will interact. We can accomplish this within a channel-manager by keeping input atoms in a queue, and output atoms in a queue. The heads of each queue interact. After reaction, if either were replicated, they are sent to the back of the queue. However, this scheme alone does not guarantee fairness across fused channel-managers: it allows two replicated terms $!\overline{x} \mid !x$ to interact indefinitely, even in the presence of a forwarder from x to y and another term $\overline{y}.P$. In this case the $\overline{y}.P$ would stay starved of interaction, even though atomic fairness requires that eventually it will interact. One way to guarantee atomic fairness is to suppress interaction when migration or deployment is possible. This then yields the same fairness property as Pierce and Turner's uniprocessor implementation [56].

It is more conventional to describe the fairness of *transitions* rather than atoms. *Strong fairness* is the property that, if it remains persistently possible for a transition to happen, then eventually it will. (Also, *weak fairness* is the property that, if it is possible infinitely often for a transition to happen, then eventually it will.) For example, consider the program $!\overline{u}.P \mid !u.Q \mid !\overline{u}.R \mid !u.S$. If every transition were stipulated as strongly fair, then each of the reactions (P, Q), (P, S), (Q, R), (R, S) must happen infinitely often. But the scheme described in the previous paragraph would only result in (P, Q) and (R, S) being executed, never (P, S) or (Q, R). It is not clear how to implement transition-based fairness in the fusion machine. It is not even clear that it should be implemented: in a real program, the programmer might wish for arbitrarily complicated fairness criteria. The implementation should provide useful and simple primitives, and leave to the programmer to implement anything more complicated.

There is more work to be done on fairness for the pi calculus. All authors on pi calculus implementation, myself included, mention it as an issue and then fail to develop any substantial theory. But it should not be so hard. The standard reference on fairness is the book *Temporal Verification of Reactive Systems* by Manna and Pnueli [39]. They describe how to turn any temporal formula—perhaps a fairness formula—into a finite transition graph. We can then cross this with the transition graph representing our program, and read fairness properties directly off the resulting graph. I leave this for future work.

Failure. Distributed systems have been jokingly defined by Lamport as ones in which 'the failure of a computer you didn't even know existed can render your own computer unusable'. We need our programs to recover gracefully in the presence of failure. There are three parts to this task: we need a model for the sort of failures that will occur in the fusion machine; we need the explicit fusion calculus to be able to express the state of the machine after failure; and we need some mechanism by which a program can be informed about the failure. The second part is already satisfied by the explicit fusion calculus, since every transition in the fusion machine corresponds to an atomic step in the calculus.

As for modelling the sort of failures that occur, this depends on the intended

use of the fusion machine. If it is used at a lower level, perhaps to implement the Internet Protocol, the typical mode of failure is that a message is lost in transit. This can happen whenever the message passes through a congested part of the network. However, most applications use the higher-level Transport Communication Protocol (TCP). This ensures reliable delivery of messages, so the typical mode of failure is for a message simply to be undeliverable: perhaps the destination machine is switched off, disconnected or crashed. If the fusion machine is used in a single machine, then message delivery is also reliable, and failure typically happens because the destination program is not expecting a message, or not executing.

The effect of message loss is clear in the machine. And at the calculus level, the effect of message loss is that a term simply vanishes. Berger and Honda [6] have shown how to recover from message loss through adding timing commands. Perhaps their techniques can be extended to the fusion machine. The effect of undeliverable messages is also clear in the machine: a transition simply does not happen. I do not know how best to recover from this at the calculus level. Perhaps each fragment in a channel should be associated with an error-handler, to be invoked upon a failed attempt to deploy or migrate that fragment.

The failure of channel-managers is also relevant for a program's *robustness*. Consider a collection of channel managers which, through their fusion pointers, denote a tree. If any single channel manager fails, then all its children will be unable to refer to the root of the tree. Perhaps this fragility could be solved by using not a tree of forwarders, but rather some structure with more redundancy. Leth and Thomsen, for instance, describe a version of Facile with multiple copies of each channel-manager [38].

Efficiency and failure-safety are in opposition. An efficient concurrent system allows many fragments to execute in parallel, even allowing subsequent commands to start executing before the first has completely finished. A failsafe system, on the other hand, must prevent subsequent operation if the first command failed. The same conflicting requirements have been encountered—and to some extent solved—in CPU design and in databases. A CPU, for instance, typically handles up to a hundred instructions at the same time, and aborts them all if an error is found. Perhaps the work in these fields might indicate how best to add efficient failure-safety to the pi calculus and explicit fusion calculus.

Conclusions. In this chapter we have seen how the fusion machine works. It uses interaction, migration and deployment: interaction to create fusions, migration to use them, and deployment to interpret commands in the calculus and also to manage trees of fusions. We have also seen how co-location allows for greater efficiency.

It may already be obvious to some readers that the fusion machine really does implement the explicit fusion calculus and pi calculus correctly, and that co-location really does help. The following chapter gives a formal proof of these two facts.

Chapter 6

Theory of fusion machine

The fusion machine is a distributed implementation of the explicit fusion calculus and of the pi calculus. This chapter proves it is a correct and efficient implementation.

The first task is to give a formal algebra for the machine. An unfortunate consequence of the formalism is that it is less approachable than the diagrams of the previous chapter, and we must spend some initial effort to prove properties about the formalism itself rather than the machine. The reward of the formalism is that, through the detailed proofs, we will learn things about the machine that are not so apparent from the diagrams: which invariants are satisfied by the tree structure; what a machine context is; how the structural congruence of the calculus is implemented in the machine. The plan of the chapter is as follows:

6.1 *Overview.* This chapter introduces some new techniques, and extends conventional techniques in new ways. The first section provides an overview of the new techniques.

6.2 *The machine calculus.* We give an algebraic description of the machine: its syntax, transitions and observation relation. We define a barbed bisimulation that is strong with respect to interaction transitions, and weak with respect to deployment and migration transitions.

6.3 *Correctness for the explicit fusion calculus.* We prove that the machine is a sound and complete ('fully abstract') implementation of the explicit fusion calculus: no matter which other programs are also executing, two programs will have the same behaviour in the machine if and only if they are barbed congruent in the calculus.

6.4 *Correctness for the pi calculus.* We prove that the machine is also a sound implementation of the pi calculus. This largely follows from the previous section and Chapter 4.

6.5 *The located machine.* In this section we augment the syntax to include information about co-location. Using this we give a *costed* reaction relation to account for the cost of each message.

6.5 *Flattening.* A program can be made efficient by fragmenting it and predeploying those fragments. A *flattening* is a complete fragmentation down

to the level of individual input and output commands. We give a flattening and prove it correct with respect to strong bisimulation congruence.

6.6 *Efficiency of flattening.* The flattening given in Section 6.5 is efficient: it requires no additional messages. We also study the efficiency of the flattening given by Laneve and Victor [34].

6.1 Overview

This chapter gives a formal calculus for the fusion machine, and proves its correctness and efficiency. Much of the work in this chapter covers new ground. I have therefore had to invent new techniques, and adapt existing techniques in unusual ways. This section outlines the techniques.

Machine calculus. The formal calculus we give for the fusion machine lies part way between the diagrams from the previous chapter, and the explicit fusion calculus. We will not prove that the fusion machine calculus corresponds to the diagrams: this should be obvious, since the two are so similar. But we will prove that the fusion machine calculus corresponds to the explicit fusion calculus; and the similarity between the two will allow for easier proofs. Let us now consider the differences between the machine and the calculus.

Restriction is the first major difference. The fusion machine itself does *not need restriction*: restriction is just a (theoretical) model for the (practical) use of fresh names. As discussed in the previous chapter (Section 5.2, page 81), most of the roles of restriction in the calculus are not needed in an implementation: there is no need for alpha-renaming, or scope-extrusion, or avoiding clashes when composing two fusion machines in parallel.

The final use of restriction is to help define an observation relation, for purposes of judging program equivalence. But here it is merely a convenience, not a necessity. In particular, while it is necessary for equivalence to have restrictions in programs, it is not necessary to have restriction in the machine contexts which surround these programs.

For consider two programs in the explicit fusion calculus $(u)(\overline{u}.\overline{x} \mid u)$ and $(v)(\overline{v}.\overline{x} \mid v)$. These are equivalent, since the channels u and v are local. Moreover, they are equivalent in all contexts, even in a context $E = \_ \mid \overline{u}$, since the context cannot react with the local channels. Were they to lack their restrictions, they would no longer be equivalent in that context. This is why restriction within a program is a necessary part of program equivalence.

However, the restriction is only necessary on the *inside* of contexts, in the programs P, and is not needed in the contexts. For let us consider one set of contexts $\mathcal{E}_r$ which has restriction, and another set $\mathcal{E}_{\not r}$ which does not. Given any program P whose contextual equivalence we are judging, and any example context $E \in \mathcal{E}_r$, we can construct an equivalent context $E' \in \mathcal{E}_{\not r}$ simply by replacing all restricted names with fresh names. Therefore, contexts $\mathcal{E}_{\not r}$ without restriction are no less discriminating than contexts $\mathcal{E}_r$ with restriction.

Although restrictions are not needed in contexts, and hence not needed in the machine, we will introduce a particular form of restriction in the fusion machine for convenience. We will write $(\!|x|\!)$ to indicate that the name x is not listed in the central registry of free names: therefore it cannot be observed,

and is not part of the observation relation. This makes for a straightforward connection with the calculus. Note that it does not allow alpha-renaming or scope extrusion like normal restriction. An alternative, avoiding all forms of restriction in the machine but at the cost of complexity, would have been to work with contextual equivalence rather than barbed congruence.

Counting transitions is the second major difference between the machine and the calculus. The transitions of the fusion machine divide into two groups, which we distinguish with the annotations $\xrightarrow{\tau}$ and $\xrightarrow{\equiv}$.

- $\xrightarrow{\tau}$, comprising the interaction transitions. These correspond to interaction steps in the calculus.
- $\xrightarrow{\equiv}$, comprising the migration and deployment transitions. These correspond to structural congruence in the calculus.

In the calculus, strong bisimulation counts the number of interaction steps but not the number of structural congruence steps. Therefore, in proving the machine correct with respect to the calculus, we introduce a form of bisimulation which counts in the same way (Section 6.4). One might say that this is a 'strong-weak' bisimulation—strong with respect to $\xrightarrow{\tau}$, and weak with respect to $\xrightarrow{\equiv}$.

But when proving efficiency properties of the machine, we will be concerned with the number of inter-location messages, not with the number of interactions. The two are almost opposites: interaction steps are all local, and structural congruence steps generally cost a message. To count the number of messages, we introduce a different annotation on transitions:

- $\longrightarrow^0$, comprising those transitions that can be accomplished without any inter-location messages.
- $\longrightarrow^1$, comprising those transitions that require an inter-location message.

The local transitions $\longrightarrow^0$ are essentially free. When proving the fusion machine efficient, we will count the number of messages using the costed transitions (Section 6.10). Strictly speaking, we should be concerned about the size as well as the number of messages. However, the efficiency results in this chapter only use fixed-sized messages, so we ignore size.

Replication is the third difference between machine and calculus. The explicit fusion calculus has arbitrary replication $!P$, where P may be term in the calculus. But this is awkward to implement. Following Pierce and Turner [56], we instead consider only guarded replication $!(\tilde{x})\mu\tilde{x}.P$.

Dangling pointers are the fourth difference. The calculus never explicitly states which channel-managers exist, but instead assumes that they all do. But in an implementation, a program could not refer by name to a remote channel-manager unless that channel-manager had already been created; and a reaction would fail unless that remote channel-manager still exists. As a simplification, we consider only *complete* machines in which all named channel-managers exist. Equivalently, we say that there are no *dangling pointers* to non-existent channel-managers. As an example, the machine $c{:}[\overline{u}]$ has a dangling pointer to u, but the machine $c{:}[\overline{u}]$, $u{:}[\,]$ has no dangling pointers.

These four points—restriction, counting transitions, replication and dangling pointers—constitute the chief differences between the explicit fusion calculus and the fusion machine calculus.

Location. We introduce a *location assumption* L, and a *costed transition relation* $L \vdash M \longrightarrow^i M'$. This means: given the assumptions L about co-location, it takes i inter-location messages for M to evolve into M'.

It is unusual to separate out the co-location information from the syntax of machines in this way. More commonly, as in Distributed Pi [57] and Located Pi [5] and the Ambient Calculus [10], the location is written as part of the syntax. For instance, $n[\overline{x}.P, y.Q]$ might indicate that terms $\overline{x}.P$ and $y.Q$ are together at location n. But the advantage of separating out the co-location information is that we do not need to modify the grammar of the calculus, do not need to invent a class of things n called 'locations', and do not need to invent new transition rules to deal with locations.

In the fusion machine, co-location never changes: if two channel-managers were co-located at the start of execution, then they will be at the end. This is represented by writing the location assumptions L outside the entire reaction $M \longrightarrow^i M'$. Additionally, so that L can remain static rather than being required to grow, we take it to be an infinite equivalence class of all potential located channel-managers—rather than a finite equivalence class of the channel-managers that have so far been created.

A simplifying factor comes from the fact that all channel-managers are created fresh and unique. This means that alpha-renaming is not used, and there is no need to worry about renaming L.

Semantics. The machine provides an operational semantics for the explicit fusion calculus and the pi calculus. That is to say: a term in the calculus is merely a string of symbols. When we say that the string of symbols is a term in the calculus, we only refer to its syntax and not to it's meaning. There are different ways to give these symbols meaning. One way is through structural congruence, a reduction relation, and barbed bisimulation. Another way is through a labelled transition system and ground bisimulation. The third way, at a lower-level, is through the transitions and equivalence of the machine—in the same way that Landin's SECD machine provides a low-level operational semantics for the lambda calculus.

This discussion may seem obvious, but it leads to an unusual form of context for the machine. We are interested in machine contexts because it is practically useful to say that two programs are equivalent in all machine contexts. But the programs that go into these contexts are terms from the calculus rather than machines. We are therefore interested in the following correctness statement: *Two programs are equivalent in all calculus contexts according to calculus semantics if and only if they are equivalent in all machine contexts according to machine semantics.*

From the programmer's perspective we are also interested in a second correctness statement: *The observations and transitions of the machine are the same as the observations and transitions of the calculus.* This means that the programmer can watch a machine execute, and relate it directly to how the program is expected to execute.

Some readers may find our treatment unfamiliar, preferring instead to invent some translation $(\cdot)^*$ from one system (the calculus) to another system (the machine) and then proving that this translation preserves bisimulation. In this case, the second correctness statement would amount to saying that the

translation has all the properties of a bisimulation. Often, one would also hope that the translation is *compositional*, so that parallel composition in the calculus corresponds to parallel composition in the machine, and prefixing in the calculus to prefixing in the machine. But prefixing does not make sense as an operation on machines. Also, this approach still leaves the problem of how to implement the translation. Note for comparison that the SECD machine does not operate upon a translation of lambda terms, but rather upon lambda terms themselves.

Contexts. We said that if two programs (as strings of symbols) are judged equivalent by the calculus semantics, then they will be behave in the same way when placed in any machine contexts. For instance, a vendor might prove a program correct, and then sell its source code. But an additional, stronger correctness property would be useful: the vendor might like to partially execute the program, and sell it in a 'ready-deployed' form. We therefore need to know not just that two programs behave the same when placed in machine contexts, but also that any deployment of the programs will also be equivalent.

It is technically awkward to express formally this additional property. We need additional machine contexts which admit machines in their holes, rather than programs. But we also need to be able to write contexts such that the context provides some of the atoms for a particular channel-manager, and the machine in its hole provides others. In effect, given the context $E_m = u{:}[B_1], \_$ and the machine $M = u{:}[B_2]$ we need $E_m[M] = u{:}[B_1, B_2]$. To express this fact, while keeping contexts compositional, we need a form of 'disassociated syntax' which allows $u{:}[B_1], u{:}[B_2] \equiv u{:}[B_1, B_2]$. This whole exercise costs substantial technical complexity for little reward. It also obscures the connection between the machine calculus and the machine diagrams. I have therefore removed it from this chapter.

Mathematically, this issue relates to the sort of congruence results we obtain. Since we have not defined contexts on machines, we cannot use the stronger reduction-closed version of congruence for the machine:

- Two machines are congruent if they have the same behaviour in all contexts; moreover, after a step of execution, the results will also be congruent.

Instead, our simplification forces us to use shallow congruence in the machine; we will relate it to shallow congruence in the explicit fusion calculus. The machine's shallow congruence is as follows:

- If two programs are equivalent then, no matter what context they are placed in, they will have the same behaviour.

6.2 The machine calculus

We assume that the set $\mathcal{N}$ of names has a total order. Let m range over $\{\mathsf{out}, \mathsf{in}\}$. Let p, q range over $\{0\} \cup \mathcal{N}$, denoting pointers which may be nil. Following Section 5.4, let P range over terms in the explicit fusion calculus which use only guarded replication: thus, all replicated terms $!P$ are assumed to have the form $!(\tilde{x})\mu\tilde{x}.P'$ with $\tilde{x} \notin \mu$. So as to make some rules simpler, we adopt the notation that u^{out} and u^{in} stand for $\overline{u}$ and u.

Definition 69 (Fusion machine) *The set $\mathcal{M}$ of fusion machines, ranged over by M, is given by*

$$\begin{array}{lcll} M & ::= & (\!|\tilde{x}|\!)C & \text{(fusion machine)} \\ C & ::= & \boldsymbol{0} \mid x_p{:}[B] \mid C, C & \text{(channel-managers)} \\ B & ::= & \boldsymbol{0} \mid \mathsf{m}\tilde{x}.P \mid !\mathsf{m}(\tilde{x}).P \mid P \mid B|B & \text{(bodies)} \end{array}$$

The *basic machine* $x_p{:}[B]$ denotes a channel-manager at channel x containing *body* B and with fusion-pointer p. The *privacy list* $(\!|\tilde{x}|\!)$ is a set of distinct channel-names which are not observable from outside the machine. We omit the fusion-pointer $x{:}[B]$ to stand for a machine with some unspecified fusion pointer: either $x_y{:}[B]$ or $x_0{:}[B]$. We write $x{:}[\,]$ for $x{:}[\mathbf{0}]$, and $\tilde{x}{:}[B]$ for $x_1{:}[B], \ldots x_n{:}[B]$. We write the empty privacy list $(\!|\,|\!)C$ as just C.

The body inside a machine is an unordered collection of *basic atoms* $\mathsf{m}\tilde{x}.P$, *replicated atoms* $!\mathsf{m}(\tilde{x}).P$ and *terms* P. In the fusion machine calculus, the body combines the 'atoms' and the 'deployment area' given in the machine diagrams from the previous chapter. Note that all fusions $x{=}y$ are already ranged over by terms P. By an abuse of notation, we also allow terms P to range over fusions $x{=}p$ that include a possible-nil name.

In the previous chapter we had assumed it possible to pick an arbitrary element from the various collections—input atoms, output atoms and deployment terms. In this chapter they are all written together in the body, and we use the pattern-matching properties of a term rewriting system to pick arbitrary elements of different types.

We make a pun between operators on terms and operators on bodies: the expression $P \mid Q$ is taken as a pair of terms in the body, rather than a single term. Similarly, in $x{:}[\mathbf{0}]$, we take the $\mathbf{0}$ to refer to a nil body rather than a nil term. As justification, we might say that all calculus terms have been pre-compiled into bodies. It would have been possible to use a separate notation $|_b$ and $\mathbf{0}_b$ for the body operators, and to have the computational steps *dep.par* and *dep.nil* as in Section 5.3 (page 83), but this would cost clarity for no gain.

Well-formedness. There are two well-formedness conditions on machines. First, recall from the previous chapter that there is exactly one channel-manager per channel. In the calculus, we say that a machine is *singly-defined* when it satisfies this condition. Formally, define

$$\operatorname{chan} x{:}[B] = x \qquad \operatorname{chan} C_1, C_2 = \operatorname{chan} C_1 \cup \operatorname{chan} C_2$$

and say that C_1, C_2 is singly-defined if and only if $\operatorname{chan} C_1 \cap \operatorname{chan} C_2 = \emptyset$.

Second, it does not make sense to write a program that refers to a machine which does not exist. Say that a machine is *complete* when it has no such 'dangling pointers'. Formally, define

$$\begin{array}{ll} \operatorname{ptr} x_y{:}[B] = y \cup \operatorname{ptr} B & \operatorname{ptr} \mathsf{m}\tilde{x}.P = \tilde{x} \cup \operatorname{fn} P \\ \operatorname{ptr} x_0{:}[B] = \operatorname{ptr} B & \operatorname{ptr} !\mathsf{m}(\tilde{x}).P = \operatorname{fn}(\tilde{x})P \\ \operatorname{ptr} C_1, C_2 = \operatorname{ptr} C_1 \cup \operatorname{ptr} C_2 & \operatorname{ptr} P = \operatorname{fn} P \\ & \operatorname{ptr} B_1|B_2 = \operatorname{ptr} B_1 \cup \operatorname{ptr} B_2 \end{array}$$

and say that a machine $(\tilde{x})C$ is complete if and only if $\mathrm{ptr}\, C \subseteq \mathrm{chan}\, C$ and $\tilde{x} \subseteq \mathrm{chan}\, C$. A machine is *well-formed* when it is both singly-defined and complete. In the following, we consider only well-formed machines.

We will use a structural congruence to identify terms that have the same physical structure: for instance, $C_1, C_2 \equiv C_2, C_1$. This is to bridge the gap between syntax and implementation: although in the implementation there is no inherent order on a collection of machines or their bodies, a linear syntax does have an order. The structural congruence abstracts away from this detail of the syntax.

Definition 70 *The* structural congruence $\equiv$ *between machines is the least congruence satisfying the following laws on channel-managers and bodies:*

1. *Abelian monoid laws with* $\boldsymbol{0}$ *as identity*
$$C, \boldsymbol{0} \equiv C \qquad C_1, C_2 \equiv C_2, C_1 \qquad C_1, (C_2, C_3) \equiv (C_1, C_2), C_3$$
$$B \mid \boldsymbol{0} \equiv B \qquad B_1 \mid B_2 \equiv B_2 \mid B_1 \qquad B_1 \mid (B_2 \mid B_3) \equiv (B_1 \mid B_2) \mid B_3$$
2. *fusion laws on atoms*
$x{=}x \equiv \boldsymbol{0} \qquad x{=}0 \equiv \boldsymbol{0} \qquad x{=}y \equiv y{=}x.$

The fusion laws have been added just as a shorthand for use in the *(dep.fu)* transition below. This transition only fuses from the lesser name to the greater, and it discards the resulting fusions if they are empty. Using the fusion laws, we will be able to write the fusion transition more simply. The fusion laws can be easily be implemented as part of that transition.

Note that $\equiv$ is not congruential with respect to structural congruence on terms in the explicit fusion calculus. That is to say: given $P \equiv Q$ in the calculus, we cannot deduce that $x{:}[P] \equiv x{:}[Q]$ as the machine. After all, the point of the machine is to show how to implement the calculus in small, easily programmable steps—including how to implement the structural congruence—and structural congruence in the calculus is not easily programmable. (It is not even known whether structural congruence is decidable. However, Engelfriet and Gelsema have at least shown for the pi calculus that structural congruence becomes decidable with the addition of some axioms [15]).

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

In giving the transition relation, we *implicitly assume further bodies in channel-managers*. In other words, we omit the bodies in channel-managers which are not changed by the presented rule. For instance, the verbose form of the *(migrate)* rule is

$$u_v{:}[\mathsf{m}\tilde{x}.P \mid B_1],\ v{:}[B_2] \longrightarrow u_v{:}[B_1],\ v{:}[\mathsf{m}\tilde{x}.P, B_2],$$

but we will present it as just

$$u_v{:}[\mathsf{m}\tilde{x}.P],\ v{:}[\,] \longrightarrow u_v{:}[\,],\ v{:}[\mathsf{m}\tilde{x}.P].$$

This convention allows us to focus on the interesting parts of the rules. It is a standard convention in work on the Join calculus [17].

Definition 71 *The transition relation $\longrightarrow$ between well-formed machines is the smallest relation satisfying the rules below, and closed with respect to structural congruence. A name is* fresh *with respect to a machine $(\tilde{x})C$ when it is not in $\tilde{x}$ or $\operatorname{chan} C$. In the following rules we take $\tilde{x}'$ and $\tilde{y}'$ to be fresh and use the abbreviations $P' = P\{\tilde{x}'/\tilde{x}\}$ and $Q' = Q\{\tilde{y}'/\tilde{y}\}$.*

$$u{:}[\mathsf{out}\tilde{x}.P \mid \mathsf{in}\tilde{y}.Q] \longrightarrow u{:}[\tilde{x}{=}\tilde{y} \mid P \mid Q] \qquad \text{(int)}$$

$$u{:}[!\mathsf{out}(\tilde{x}).P \mid \mathsf{in}\tilde{y}.Q] \longrightarrow (\tilde{x}')\ u{:}[\tilde{x}'{=}\tilde{y} \mid P' \mid Q \mid !\mathsf{out}(\tilde{x}).P],\ \tilde{x}'_0{:}[\,] \qquad \text{(int.rout)}$$

$$u{:}[\mathsf{out}\tilde{x}.P \mid !\mathsf{in}(\tilde{y}).Q] \longrightarrow (\tilde{y}')\ u{:}[\tilde{x}{=}\tilde{y}' \mid P \mid Q' \mid !\mathsf{in}(\tilde{y}).Q],\ \tilde{y}'_0{:}[\,] \qquad \text{(int.rin)}$$

$$u{:}[!\mathsf{out}(\tilde{x}).P \mid !\mathsf{in}(\tilde{y}).Q] \longrightarrow (\tilde{x}'\tilde{y}')\ u{:}[\tilde{x}'{=}\tilde{y}' \mid P' \mid Q' \mid !\mathsf{out}(\tilde{x}).P \mid !\mathsf{in}(\tilde{y}).Q],\ \tilde{x}'_0\tilde{y}'_0{:}[\,] \qquad \text{(int.both)}$$

$$u_v{:}[\mathsf{m}\tilde{x}.P],\ v{:}[\,] \longrightarrow u_v{:}[\,],\ v{:}[\mathsf{m}\tilde{x}.P] \qquad \text{(migrate)}$$

$$u_v{:}[!\mathsf{m}(\tilde{x}).P],\ v{:}[\,] \longrightarrow u_v{:}[\,],\ v{:}[!\mathsf{m}(\tilde{x}).P] \qquad \text{(migrate.rep)}$$

$$u{:}[x{=}y],\ x_q{:}[\,] \longrightarrow u{:}[\,]\ x_y{:}[y{=}q], \quad \textit{if } x < y \qquad \text{(dep.fu)}$$

$$u{:}[(\tilde{x})P] \longrightarrow (\tilde{x}')\ u{:}[P'],\ \tilde{x}'{:}[\,] \qquad \text{(dep.new)}$$

$$u{:}[v^{\mathsf{m}}\tilde{x}.P],\ v{:}[\,] \longrightarrow u{:}[\,],\ v{:}[\mathsf{m}\tilde{x}.P] \qquad \text{(dep.action)}$$

$$u{:}[!v^{\mathsf{m}}(\tilde{x}).P],\ v{:}[\,] \longrightarrow u{:}[\,],\ v{:}[!\mathsf{m}(\tilde{x}).P] \qquad \text{(dep.rep.action)}$$

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

$$\frac{C \longrightarrow (\tilde{x})\ C' \qquad \operatorname{chan} C_2 \cap \operatorname{chan} C' = \emptyset}{(\tilde{y})\ C, C_2 \longrightarrow (\tilde{x}\tilde{y})\ C', C_2}$$

Let $\xrightarrow{\tau}$ range over (int) rules, and $\xrightarrow{\equiv}$ over the others.

All transition rules preserve well-formedness. In respect of this, note that *(dep.new)* and the replicated interaction rules create fresh, empty channel-managers so as to preserve completeness. Meanwhile, the side-condition on the context closure rule ensures that all names are singly-defined—in other words, the freshly created names will be globally unique. This is different from context-closure in explicit fusion calculus, where the names are local but not necessarily unique. This is why the explicit fusion calculus needs alpha-renaming, but the fusion machine does not.

The rules have all been explained in the previous chapter. The most subtle rule is *(dep.fu)*. As was explained, this rule leads to a tree structure of fusion pointers. We now explain the tree structure formally. To this end we use two relations concerning the trees. First, $x \rightsquigarrow y$ means that there is a forwarding path from x to y. Second, $x \stackrel{\rightsquigarrow}{=} y$ means that x and y are on the same tree: hence, two complementary atoms at x and y respectively can migrate to a common name z and then react.

Definition 72 *The relation $\rightsquigarrow$ on names, for a given machine $(\tilde{x})C$, is the least transitive relation satisfying*

- *$x_y{:}[B] \in C$ implies $x \rightsquigarrow y$.*

The relation $\stackrel{\rightsquigarrow}{=}$ on names is the least relation on names satisfying

- $x \overset{\rightsquigarrow}{=} y$ *if there exists a* z *such that* $x \rightsquigarrow z \vee x = z$, *and* $y \rightsquigarrow z \vee y = z$.

Note that this relation is concerned only with the structure of the tree inside the machine, regardless of which names $(\!|\tilde{x}|\!)$ are hidden from observers. Clearly, $x \rightsquigarrow y$ if and only if atoms can migrate from x to y. The relation $\rightsquigarrow$ also has a correctness invariant. Basically, the relation is a tree (i.e. no loops, no divergence) which respects the total order $>$ on names. Formally:

Lemma 73 *The following properties hold for a machine* $c{:}[P]$ *where* c *is any name and* P *is any program. The properties are also preserved by transitions.*

1. *(Anti-reflexive) For no* x *does* $x \rightsquigarrow x$.
2. *(Anti-symmetric)* $x \rightsquigarrow y$ *and* $y \rightsquigarrow x$ *implies* $x = y$.
3. *(Transitive)* $x \rightsquigarrow y$ *and* $y \rightsquigarrow z$ *implies* $x \rightsquigarrow z$.
4. *(Confluent)* $x \rightsquigarrow y$ *and* $x \rightsquigarrow z$ *implies* $y \rightsquigarrow z$ *or* $z \rightsquigarrow y$ *or* $y = z$.
5. *(Order-respecting)* $x \rightsquigarrow y$ *implies* $x < y$.

Proof. Note that the only transition to modify fusion pointers, and hence the only transition relevant to this proof, is *(dep.fu)*. The side condition that $x < y$ means that $x \neq y$, so the result of the transition is anti-reflexive. It also means that $y \not\rightsquigarrow x$, so adding $x \rightsquigarrow y$ does not break anti-symmetry. It also means that $x \rightsquigarrow y$ is order-respecting. And transitivity and confluence are straightforward. □

We will find it helpful to consider 'fully-deployed' machines, in which all terms P inside the bodies have been fully deployed into atoms. This result is a convenience: with it we will be able to analyse solely the tree structure, without having to worry about fusions inside bodies.

Lemma 74 *Given some machine* M, *there exists another machine* M' *such that* $M \overset{\equiv}{\longrightarrow}^{*} M'$ *and* M' *contains only atoms inside its bodies; no terms.*

Proof. Consider the total size of all terms inside all bodies. All of the deployment transitions apart from (dep.fu) strictly decrease the total size of terms in the body. As for (dep.fu), it keeps the size constant until eventually it reaches an un-fused machine, at which point it decreases the size. This will necessarily eventually happen, since machines are finite and, by the anti-symmetry property, there are no loops in the tree of fusion pointers. Finally, every term P in a body admits at least one deployment transition. □

6.3 Observation relation

We now introduce the observation relation on machines, and show that an atom at some channel x may be observed at every $y \overset{\rightsquigarrow}{=} x$ so long as y is not private.

Definition 75 *The observation on machines is*

$$(\!|\tilde{x}|\!)C \xrightarrow{\mu} \text{ if } \tilde{x} \notin \mu \text{ and } C \xrightarrow{\mu},$$

where observation $C \xrightarrow{\mu}$ *on channel-managers is*

$$\begin{aligned} u{:}[\mathsf{out}\tilde{x}.P \mid B] &\xrightarrow{\overline{u}} \\ u{:}[\mathsf{in}\tilde{x}.P \mid B] &\xrightarrow{u} \\ u{:}[!\mathsf{out}(\tilde{x}).P \mid B] &\xrightarrow{\overline{u}} \\ u{:}[!\mathsf{in}(\tilde{x}).P \mid B] &\xrightarrow{u} \end{aligned}$$

$$\begin{aligned} u_v{:}[B], C &\xrightarrow{\overline{u}} \ \textit{if } C \xrightarrow{\overline{v}} \\ u_v{:}[B], C &\xrightarrow{u} \ \textit{if } C \xrightarrow{v} \\ C_1, C_2 &\xrightarrow{\mu} \ \textit{if } C_1 \xrightarrow{\mu} \textit{ or } C_2 \xrightarrow{\mu} \\ u{:}[B] &\xrightarrow{\mu} \ \textit{if } B \equiv B' \textit{ and } u{:}[B'] \xrightarrow{\mu} \end{aligned}$$

The principle behind these relations is as follows. To make an observation $\xrightarrow{u}$, the observer notionally places a term $\overline{u}$ into the machine, and tests whether the term reacts. So that the observer knows about the name u, it cannot be in the private list. Once the test-term has been placed, there are three possibilities:

- *(In place)* Perhaps the machine at u already contains an input atom. Then this can react immediately, and this reaction can be observed.
- *(Down)* Perhaps there is some machine $v{:}[\mathsf{in}]$ with $v \rightsquigarrow u$. Then this atom at v could migrate to u, and the ensuing reaction could be observed. We will account for this through a form of weak bisimulation: to match one machine's observation, another machine is allowed to first perform some migration steps.
- *(Up)* Perhaps there is some machine $v{:}[\mathsf{in}]$ with $u \rightsquigarrow v$. Then the observe's test atom could migrate from u to v, and the ensuing reaction could be observed. We account for this through the rules $u_v{:}[B], C \xrightarrow{u}$ if $C \xrightarrow{v}$.

The following lemma characterises structurally the possible causes of an observation $C \xrightarrow{\overline{u}}$.

Lemma 76 *If a collection of channel-managers does* $\xrightarrow{\overline{u}}$*, then the collection is in fact one of the following.*

$$\begin{aligned} u{:}[B] \quad & \textit{with} \quad B \equiv \mathsf{out}\tilde{x}.P \mid B' \\ u{:}[B] \quad & \textit{with} \quad B \equiv !\mathsf{out}(\tilde{x}).P \mid B' \\ C_1, C_2 \quad & \textit{with} \quad C_1 \xrightarrow{\overline{u}} \textit{ or } C_2 \xrightarrow{\overline{u}} \\ u_v{:}[B], C \quad & \textit{with} \quad C \xrightarrow{\overline{v}} \end{aligned}$$

Similarly for $\xrightarrow{u}$*.*

Proof. Note that the observation relation was defined inductively on the structure of machines: that is, although it involves a structural congruence on bodies, it does not involve a structural congruence on machines. This means that a straightforward induction on the derivation of $C \xrightarrow{\overline{u}}$ suffices to prove the lemma. □

The following lemma relates the tree-structure of a machine, to its observations. We will use this result in the following section to prove the machine's correctness.

Lemma 77 *If* $C \xrightarrow{\overline{x}}$ *and* $x \overset{\rightsquigarrow}{=} y$, *then* $C \xrightarrow{\equiv}^{*} \xrightarrow{\overline{y}}$.

Proof. If $C \xrightarrow{\overline{x}}$ then (by Lemma 76) C contains some atom $u\,[\mathsf{out}\tilde{z}.P]$, perhaps replicated, with $x \rightsquigarrow u$ or $x = u$. We will consider the more interesting case that $x \neq u$. We are given that $x \overset{\rightsquigarrow}{=} y$: in other words, there exists some z such that $y \rightsquigarrow z$ and also $x \rightsquigarrow z$. But also $x \rightsquigarrow u$. From the confluence property (Lemma 73), there are three possibilities.

1. Perhaps $z \rightsquigarrow u$. We also have $y \rightsquigarrow z$; hence, by transitivity, $y \rightsquigarrow u$. Therefore $C \xrightarrow{y}$.
2. Perhaps $u \rightsquigarrow z$. Therefore, C can undergo some migrations such that the atom in ends up at z. But since we have $y \rightsquigarrow z$, we get $C \xrightarrow{\equiv}^{*} \xrightarrow{y}$.
3. Perhaps $z = u$. From $y \rightsquigarrow z$ we deduce $y \rightsquigarrow u$. Therefore $C \xrightarrow{y}$. □

6.4 Machine bisimulation

We now define bisimulation on machines. We also define the circumstances under which two programs are judged equivalent in the machine.

As discussed in the introduction, part of our goal is to show how the machine corresponds to the explicit fusion calculus. Now the machine's $\xrightarrow{\equiv}$ transitions show explicitly how to accomplish structural congruence, while the calculus simply assumes structural congruence. To prove the connection, we must therefore not observe the $\xrightarrow{\equiv}$ transitions. As before, we write $\xrightarrow{\equiv}^{*}$ to stand for a sequence of zero or more $\xrightarrow{\equiv}$.

Definition 78 *A barbed bisimulation* $\mathcal{S}$ *between machines is the least relation such that if* $M\ \mathcal{S}\ N$ *then*

- $M \xrightarrow{\mu}$ *implies* $N \xrightarrow{\equiv}^{*} \xrightarrow{\mu}$
- $N \xrightarrow{\mu}$ *implies* $M \xrightarrow{\equiv}^{*} \xrightarrow{\mu}$
- $M \xrightarrow{\equiv}^{*} \xrightarrow{\tau} M'$ *implies there exists* N' *such that* $N \xrightarrow{\equiv}^{*} \xrightarrow{\tau} \xrightarrow{\equiv}^{*} N'$ *and* $M'\ \mathcal{S}\ N'$
- $N \xrightarrow{\equiv}^{*} \xrightarrow{\tau} N'$ *implies there exists* M' *such that* $M \xrightarrow{\equiv}^{*} \xrightarrow{\tau} \xrightarrow{\equiv}^{*} M'$ *and* $M'\ \mathcal{S}\ N'$

Let $\dot{\sim}_b$, called *barbed bisimulation*, be the largest barbed bisimulation. It is easy to check that $\dot{\sim}_b$ is an equivalence relation.

We now give contexts for the machine. Recall that E_ϕ ranges over explicit fusion contexts (Definition 2, page 23). We will define E_m to range over machine contexts. Note that the holes in these machine contexts admit calculus programs, not other machines. That is because we are interested in how programs behave when placed in a machine, not how one machine behaves when placed next to another machine.

Definition 79 *The set $\mathcal{E}_m$ of machine contexts is given by*

$$\begin{array}{rcl} E_m & ::= & (\tilde{x})E_c \\ E_c & ::= & x_p{:}[E_b] \mid C, E_c \mid E_c, C \\ E_b & ::= & \mathsf{m}\tilde{x}.E_\phi \mid !\mathsf{m}(\tilde{x}).E_\phi \mid E_\phi \mid B|E_b \mid E_b|B \end{array}$$

Again, we assume that E_ϕ uses only guarded replication. When we write a machine $E_m[P]$, we implicitly assume it to be well-formed. When we write a program in a basic context $c_0{:}[P]$, it is shorthand for the (complete) machine $c_0{:}[P]$, $\tilde{x}{:}[\mathbf{0}]$ where $\tilde{x} = \mathrm{fn}(P)\backslash c$.

We will say that the fusion machine semantics judge two programs equivalent if those programs give rise to bisimilar machines when placed in any context.

Definition 80 (Machine equivalence) *The relation $\sim_m$ between programs is $P \sim_m Q$ iff $\forall E_m : E_m[P] \dot{\sim}_b E_m[Q]$.*

This machine equivalence is basically a shallow barbed congruence.

6.5 Correctness for the explicit fusion calculus

In this section we prove that the fusion machine is a correct implementation of the explicit fusion calculus.

The first part is to prove that a term has the same barbs and transitions when executed on a machine, as it does in the calculus: if $P \xrightarrow{\mu}$ in the calculus, then $c_0{:}[P] \xrightarrow{\mu}$ in the machine, and similarly for tau transitions. In practical terms, this means that any two programs judged to be barbed bisimilar in the calculus, will have the same behaviour when run (in isolation) on the machine.

The second part is to show that the machine semantics make the same judgements about program equivalence, as do the explicit fusion calculus semantics: P and Q are shallow barbed congruent in the calculus if and only if $P \sim_m Q$. In practical terms, any two programs judged to be congruent in the calculus, will have the same behaviour when run within any context on the machine.

(Note that in Chapter 3, by contrast, we focused primarily on the stronger reduction-closed barbed congruence. Recall from Section 6.1, however, that the contexts we use in this chapter cannot be used to define reduction-closed congruence. That is why this chapter uses shallow congruence.)

The proof works as follows. We are given some term P, and we consider the reactions it can undergo. We must then show that the corresponding machine can also undergo equivalent transitions. However, the task is a made complex in two ways. First, there are more execution paths possible in the machine than the calculus, involving more transitions. Second, a single term in the calculus might be represented in multiple ways by different machines: for instance, $x{=}y$ corresponds to $x_y{:}[\,]$ as well as $u_0{:}[x{=}y]$.

There is a simple solution to this complexity: we give a translation from machines into terms. In effect, we translate from the lower-level language (the machines) into the higher-level language (the explicit fusion calculus). This might seem surprising—after all, and in contrast, in Section 4.5 we translated from the higher-level (pi calculus) to the lower (explicit fusion calculus). But,

just as in that section, we will prove that the translation preserves barbs and tau transitions in both directions, and so the direction of the translation does not matter. Turner [68] also used a reverse translation to prove his uniprocessor machine correct.

We actually use two subsidiary translation functions: one to translate a machine into the calculus, keeping a note of all private names; and one to translate bodies into the calculus.

Definition 81 *The translation* calc *from machines to terms is*

$$\mathrm{calc}(\!|\tilde{x}|\!)C \;=\; (\tilde{x})\,\mathrm{calc}\,C$$

with the subsidiary translation calc *from channel-managers to terms:*

$$\begin{aligned}
\mathrm{calc}\,\boldsymbol{0} &= \boldsymbol{0}\\
\mathrm{calc}\,u_v{:}[B] &= u{=}v \mid \mathrm{calc}_u\,B\\
\mathrm{calc}\,u_0{:}[B] &= \mathrm{calc}_u\,B\\
\mathrm{calc}\,C_1, C_2 &= \mathrm{calc}\,C_1 \mid \mathrm{calc}\,C_2
\end{aligned}$$

and calc_u *from bodies to terms, parameterised on the name* u*:*

$$\begin{aligned}
\mathrm{calc}_u\,\boldsymbol{0} &= \boldsymbol{0}\\
\mathrm{calc}_u\,\mathsf{m}\tilde{x}.P &= u^{\mathsf{m}}\tilde{x}.P\\
\mathrm{calc}_u\,!\mathsf{m}(\tilde{x}).P &= !(\tilde{x}')u^{\mathsf{m}}\tilde{x}'.P\{\tilde{x}'/\tilde{x}\}, \quad u \notin \tilde{x}',\ \tilde{x}' \textit{ distinct, fresh}\\
\mathrm{calc}_u\,P &= P\\
\mathrm{calc}_u\,B_1|B_2 &= \mathrm{calc}_u\,B_1 \mid \mathrm{calc}_u\,B_2
\end{aligned}$$

We now prove that $\xrightarrow{\mu}$ observations and $\xrightarrow{\tau}$ transitions are the same between the calculus and the machine. This involves the subsidiary lemmas: first that $\xrightarrow{\equiv}$ transitions in the machine really do correspond to $\equiv$ in the calculus; and second that barbs and observation are preserved.

Lemma 82

1. $M \equiv N$ *implies* $\mathrm{calc}\,M \equiv \mathrm{calc}\,N$.
2. $M \xrightarrow{\equiv} M'$ *implies* $\mathrm{calc}\,M \equiv \mathrm{calc}\,M'$.

Proof. The first part is an induction on the derivation of $M \equiv N$. It uses the auxiliary result that $B_1 \equiv B_2$ implies $\forall x.\ \mathrm{calc}_x\,B_1 \equiv \mathrm{calc}_x\,B_2$. The second part is an induction on the derivation of the transition. It uses the fact that if $x \notin \mathrm{fn}(B)$ and $x \neq u$ then $x \notin \mathrm{fn}(\mathrm{calc}_u\,B)$. □

Lemma 83

1. $M \xrightarrow{\mu}$ *implies* $\mathrm{calc}\,M \xrightarrow{\mu}$
2. $M \xrightarrow{\tau} M'$ *implies* $\mathrm{calc}\,M \xrightarrow{\tau} \mathrm{calc}\,M'$

Proof. The first part is proved by a straightforward induction on the structure of M using Lemma 76. We use the fact that $P \xrightarrow{v}$ implies $u{=}v \mid P \xrightarrow{u}$. For the second part, we use a straightforward induction on the derivation of $M \xrightarrow{\tau} M'$. There are four cases: interaction, equivalence (handled by Lemma 82), congruence M, N and congruence $(x)M$. □

Lemma 84

1. $\mathrm{calc}\, M \xrightarrow{\mu}$ *implies* $M \xrightarrow{\equiv}{}^{*} \xrightarrow{\mu}$
2. $\mathrm{calc}\, M \xrightarrow{\tau} P'$ *implies there exists* M' *such that* $M \xrightarrow{\equiv}{}^{*} \xrightarrow{\tau} M'$ *and* $P' \equiv \mathrm{calc}\, M'$

Proof. For the first part, suppose $\mathrm{calc}\, M \xrightarrow{u}$. There is (Lemma 74) an M' containing no terms in any of its bodies, and with all its restrictions pushed fully out, such that $M \xrightarrow{\equiv}{}^{*} M'$. Therefore (Lemma 82) $\mathrm{calc}\, M \equiv \mathrm{calc}\, M'$, and so $\mathrm{calc}\, M' \xrightarrow{\mu}$. Now $\mathrm{calc}\, M'$ has the form $(\tilde{z})P$, where all names $\tilde{z}$ are private in the machine, and P contains only fusions and actions in direct correspondence to the machine (Definition 81). If $\mathrm{calc}\, M' \xrightarrow{u}$, this must have come from some action $v\tilde{z}.Q$ with $P \vdash u{=}v$ (Proposition 25). This in turn must have come from a channel-manager $v{:}[\mathsf{in}\tilde{z}.Q]$ with $u \mathrel{\tilde{=}} v$. Therefore (Lemma 77) $M' \xrightarrow{u}$.

For the second part, we again produce an M' which contains no terms in any of its bodies such that $M \xrightarrow{\equiv}{}^{*} M'$. Since $\mathrm{calc}\, M \xrightarrow{\tau} P'$, and $\mathrm{calc}\, M \equiv \mathrm{calc}\, M'$, it must be that $\mathrm{calc}\, M' \xrightarrow{\tau} P'$. Again, consider the form of $\mathrm{calc}\, M'$. It must have some action $\overline{u}\tilde{x}.Q_1$, and another action $v\tilde{y}.Q_2$, such that u and v are related through fusions. Therefore, M' also has those actions, and $u \mathrel{\tilde{=}} v$. Therefore it is possible for the atoms in M' to migrate to a common point and react as desired. □

Corollary 85 (Bisimulation)

1. $M \mathrel{\dot{\sim}_b} N$ *if and only if* $\mathrm{calc}\, M \mathrel{\dot{\sim}_b} \mathrm{calc}\, N$.
2. $P \mathrel{\dot{\sim}_b} Q$ *if and only if* $c_0{:}[P] \mathrel{\dot{\sim}_b} c_0{:}[Q]$.

We now extend the result to congruence: machine equivalence is the same as shallow barbed congruence in the calculus.

Let us first consider how this congruence result differs from the bisimulation result (Corollary 85), and why it is necessary before we can claim that the fusion machine is correct. The bisimulation result means that, for any programs running in isolation, they will have the same behaviour in the machine as they do in the calculus. But we are really concerned with interactive systems. When we deploy a program onto a system, the rest of the system—the context—may be anything, and may interact with our program. We therefore need to prove that two programs that are barbed bisimilar in all calculus contexts will still be barbed bisimilar in all machine contexts. This is the soundness property.

For practical purposes, soundness is all that is needed. That said, it is also easy to prove completeness for the fusion machine. By contrast, for the Pict machine, only the soundness result is possible [62]. That is because Pict commits to a single reduction strategy.

Recall that E_m ranges over contexts in the machine (Definition 79, page 105), and that E_ϕ ranges over contexts in the explicit fusion calculus (Definition 2, page 23).

Lemma 86 (Contexts)

1. *For all E_m there exists an E_ϕ such that* $\forall P.\ \mathrm{calc}\, E_m[P] \equiv E_\phi[P]$.
2. *For all E_ϕ there exists an E_m such that* $\forall P.\ \mathrm{calc}\, E_m[P] \equiv E_\phi[P]$.

Proof. For the first part, we extend the translation calc (Definition 81) to translate contexts as well as terms, and we pick E_ϕ such that $E_m = \mathrm{calc}\, E_\phi$. The context translation is

$$\begin{aligned}
\mathrm{calc}(\tilde{x})E_c &= (\tilde{x})\,\mathrm{calc}\, E_c \\
\\
\mathrm{calc}\, u_v{:}[E_b] &= u{=}v \mid \mathrm{calc}_u\, E_b \\
\mathrm{calc}\, u_0{:}[E_b] &= \mathrm{calc}_u\, E_b \\
\mathrm{calc}\, M, E_c &= \mathrm{calc}\, M \mid \mathrm{calc}\, E_c \\
\mathrm{calc}\, E_c, M &= \mathrm{calc}\, E_c \mid \mathrm{calc}\, M \\
\\
\mathrm{calc}_u\, \mathsf{m}\tilde{x}.E_\phi &= u^{\mathsf{m}}\tilde{x}.E_\phi \\
\mathrm{calc}_u\, !\mathsf{m}(\tilde{x}).E_\phi &= !(\tilde{x}')u^{\mathsf{m}}\tilde{x}'.E_\phi\{\tilde{x}'/\tilde{x}\} \quad u \notin \tilde{x}',\ \tilde{x}' \text{ distinct, fresh} \\
\mathrm{calc}_u\, E_\phi &= E_\phi \\
\mathrm{calc}_u\, B|E_b &= \mathrm{calc}_u\, B \mid \mathrm{calc}_u\, E_b \\
\mathrm{calc}_u\, E_b|B &= \mathrm{calc}_u\, E_b \mid \mathrm{calc}_u\, B
\end{aligned}$$

It is easy to see that this satisfies the lemma.

The second part is trivial: given a context E_ϕ in the explicit fusion calculus, construct the machine context $c{:}[E_\phi]$. □

Finally we prove the congruence result. Note that we are using shallow congruence.

Proposition 87 (Congruence) $\forall E_\phi : E_\phi[P] \dot{\sim}_b E_\phi[Q]$ *iff* $P \sim_m Q$.

Proof. Follows immediately from Corollary 85 and Lemma 86. □

6.6 Correctness for the pi calculus

We now address the same results for implementing the pi calculus on the fusion machine. We will do this via the translation from the pi calculus into the explicit fusion calculus given in Section 4.5. We proved in that section that the translation into the explicit fusion calculus is sound. And we have also proved in the previous section that the explicit fusion is soundly implemented by the machine. All that remains is a definition of piability for the fusion machine. It is substantially the same as piability for the explicit fusion calculus (Definition 57, page 69).

Definition 88 (Piability) *A machine M is piable if* $\mathrm{calc}\, M$ *is piable. A machine context E_m is piable if there exists a context E_π in the pi calculus such that for all Q_π, $E_\pi[Q_\pi]^* \equiv \mathrm{calc}\, E_m[Q_\pi^*]$.*

Piability is preserved by the operations of the machine. This follows immediately from earlier results:

- Piability is preserved by $\equiv$ in the machine, since this corresponds to $\equiv$ in the explicit fusion calculus (Lemma 82) which itself preserves piability by definition.
- Piability is preserved by $\xrightarrow{\equiv}$ in the machine, for the same reason.
- Piability is preserved by $\xrightarrow{\tau}$ in the machine, since it corresponds to $\longrightarrow$ in the explicit fusion calculus (Lemma 83), which itself preserves piability (Lemma 64).

The following lemma states formally the piability properties of the machine.

Lemma 89 *Let M and M' range over piable machines, and P and P' over terms in the pi calculus.*

1. $M \xrightarrow{\mu}$ *and* $\operatorname{calc} M = P^*$ *implies* $P \xrightarrow{\mu}$
2. $P \xrightarrow{\mu}$ *implies* $c{:}[P^*] \xrightarrow{\equiv}{}^* \xrightarrow{\mu}$
3. $M \xrightarrow{\tau} M'$ *and* $\operatorname{calc} M = P^*$ *implies there exists a P' such that* $P \xrightarrow{\tau} P'$ *and* $\operatorname{calc} M' = P'^*$
4. $P \xrightarrow{\tau} P'$ *implies there exists an M' such that* $c{:}[P^*] \xrightarrow{\equiv}{}^* \xrightarrow{\tau} M'$ *and* $\operatorname{calc} M' \equiv c{:}[P'^*]$

Proof. By Lemmas 82 and 63, machine piability is closed under structural congruence and $\xrightarrow{\equiv}$ transitions; as explained above, it is also preserved by $\xrightarrow{\tau}$ transitions. We can therefore use Lemma 65 to relate all observations and tau transitions between the explicit fusion calculus to the pi calculus. □

Corollary 90 (Bisimulation) *Let P and Q be any terms in the pi calculus, and c be any location. Then $P \dot{\sim}_b Q$ if and only if $c{:}[P^*] \dot{\sim}_b c{:}[Q^*]$.*

We now prove soundness. Recall that E_π ranges over contexts in the pi calculus (Definition 53, page 68).

Proposition 91 (Soundness) $\forall E_\pi : E_\pi[Q_\pi] \dot{\sim}_b E_\pi[Q'_\pi]$ *implies* $\forall E_m piable : E_m[Q^*_\pi] \dot{\sim}_b E_m[Q'^*_\pi]$

Proof. We are given that

$$\forall E_\pi : E_\pi[Q_\pi] \dot{\sim}_b E_\pi[Q'_\pi].$$

By Corollary 66,

$$\forall E_\pi : E_\pi[Q_\pi]^* \dot{\sim}_b E_\pi[Q'_\pi]^*.$$

Now by the definition of piability, for all piable E_m,

$$\exists E_\pi : E_\pi[Q_\pi]^* \equiv \operatorname{calc} E_m[Q^*_\pi] \quad \wedge \quad E_\pi[Q'_\pi]^* \equiv \operatorname{calc} E_m[Q'^*_\pi].$$

Combining this with the previous equation we get

$$\forall E_m \text{piable} : \text{calc}\, E_m[Q^*_\pi] \equiv E_\pi[Q_\pi]^* \mathrel{\dot\sim_b} E_\pi[Q'_\pi]^* \equiv \text{calc}\, E_m[Q'^*_\pi].$$

Finally, by Corollary 85,

$$\forall E_m \text{piable} : E_m[Q^*_\pi] \mathrel{\dot\sim_b} E_m[Q'^*_\pi]. \qquad \square$$

As in Section 4.5, I further conjecture that piable machine contexts are complete as well as sound.

Conclusion. This concludes our treatment of the correctness of the fusion machine. We have shown that it is a sound and complete implementation of the explicit fusion calculus: no matter what other programs are also running on the machine (i.e. in all contexts), two programs on the machine will have the same behaviour if and only if they are shallow barbed congruent in the explicit fusion calculus.

We have also shown that the machine is a sound implementation of the pi calculus. This result assumes that all the other programs running on the machine were written in the pi calculus (i.e. they form piable contexts).

The remainder of this chapter addresses the efficiency of the fusion machine.

6.7 The located machine

We now add co-location to the fusion machine formalism, as proposed in the previous chapter (Section 5.5, page 88). That section has already explained how co-location works. The goal of the current section is to show how to define it formally, and to prepare for Section 6.10 where we use it to prove an efficiency result.

To concentrate on the important features of co-location, we make some simplifying assumptions. First, we remove replication: it is in any case orthogonal to co-location. Second, for the commands used to create fresh names, we only use located bound input commands $u(x@).P$ and not located bound output $\overline{u}(x@).P$. This halves our workload.

Note that both free names and bound names may be co-located. However, the only way for a program to *make* one name co-located with another is to create it fresh.

We will annotate the machine transitions to indicate their cost. Let L be an assumption about co-location, in a sense to be defined. We will write $L \vdash (M \longrightarrow^i M')$ to mean that, given the assumption L, it will take i inter-location messages for M to evolve into M'. For instance, using the optimised migration rule discussed in the previous chapter

$$u@v \;\vdash\; u_v{:}[\mathsf{in}] \longrightarrow^0 u_v{:}[\mathbf{0}],\; v{:}[\mathsf{in}]$$

where $u@v$ denotes that u and v are co-located.

Locations in the fusion machine never change: if one channel-manager was initially co-located with another, then it will always be. It is always possible to pick a fresh name at a fresh location; and it is always possible to pick a fresh name at an existing location. We represent this by taking L to be an

equivalence relation on names such that every equivalence class is infinite, and there are infinitely many different classes. All names within an equivalence class are physically adjacent. We write $u@v$ when u and v are in the same equivalence class.

The program $(x)P$ creates a fresh name x' that is not co-located with anything else. It does this by picking x' from one of the so-far unused equivalence classes in L. There are infinitely many equivalence classes, so that arbitrarily many fresh names can be created. In the same way, the program $(x@y)P$ picks its fresh name x' from the same equivalence class as y.

As indicated, we must extend the explicit calculus to include commands such as $(x@y)P$ for creating fresh names at specified locations. With an abuse of notation, we still use P to address terms of this extended calculus.

Definition 92 (Located calculus) *The set of terms $\mathcal{P}_\pi$ of terms in the located explicit fusion calculus is as in Definition 1, minus replication, and with two additions:*

$$P \quad ::= \quad (x@y)P \quad \mid \quad u(\tilde{x}@).P \quad \mid \quad \ldots$$

The meaning of the new operations is as follows. The *locator* $(x@y)P$ is a form of restriction which declares that the bound name x is at the same location as y. The *bound input* command $u(\tilde{x}@).P$, when it reacts with $\overline{u}\tilde{y}.Q$, will create fresh channels $\tilde{x}$ at the same location as $\tilde{y}$. In other respects, the commands behave like normal restriction and bound input.

The terminology and notation for locators is as follows. In the locator $(x@y)$, y is free. Alpha-renaming the locator $(x@y)P$ changes it to $(x'@y)P\{x'/x\}$ without modifying y. We use the abbreviation $(\tilde{x}@\tilde{y}) = (x_1@y_1)\ldots(x_n@y_n)$. Let x_l range over restrictions and locators; correspondingly, $\tilde{x}_l$ over lists of restrictions and locators. Two locators $x@y$ and $u@v$ are *distinct* if $x \neq u$, without regard to y or v. We say that $z \in \tilde{x}@\tilde{y}$ when $z \in \tilde{x}$.

Correspondingly, we also extend bodies of fusion machines with locators and located inputs. Again, by an abuse of notation, the extended bodies are addressed by B.

Definition 93 (Located bodies) *Located bodies B are as in Definition 69, without replication, and with one addition:*

$$B \quad ::= \quad \mathsf{in}(\tilde{x}@).P \quad \mid \quad \ldots$$

The *bound input atom* $\mathsf{in}(\tilde{x}@).P$, when it reacts with $\mathsf{out}(\tilde{y}).Q$, will create fresh channels $\tilde{x}$ at the same locations as $\tilde{y}$.

The transition relation is given below. The key features to note are that all transitions are divided into *remote transitions* $\longrightarrow^1$, each of which takes a single message to accomplish, and *local transitions* $\longrightarrow^0$ which take none. The rules themselves are exactly the same as for the normal fusion machine (Definition 71, page 101), except they have been classed as either remote or local. We give some example transitions after the definition.

Definition 94 *The costed transition relation $L \vdash M \longrightarrow^i M'$ is the smallest relation satisfying the rules below, and closed with respect to structural congruence. In the (dep.new) rule we assume that $\forall y \in \mathrm{fn}\, P : (x@y) \notin L$.*

Remote transitions: ($u@v \notin L$)

$$L \vdash u_v{:}[\mathsf{m}\tilde{x}.P],\ v{:}[\,] \longrightarrow^1 u_v{:}[\,],\ v{:}[\mathsf{m}\tilde{x}.P] \qquad \text{(migrate)}$$
$$L \vdash u{:}[v{=}y],\ v_p{:}[\,] \longrightarrow^1 u{:}[\,],\ v_y{:}[y{=}p], \quad \textit{if } v < y \qquad \text{(dep.fu)}$$
$$L \vdash u{:}[v^{\mathsf{m}}\tilde{x}.P],\ v{:}[\,] \longrightarrow^1 u{:}[\,],\ v{:}[\mathsf{m}\tilde{x}.P] \qquad \text{(dep.act)}$$
$$L \vdash u{:}[v^{\mathsf{m}}(\tilde{x}@).P], v{:}[\,] \longrightarrow^1 u{:}[\,],\ v{:}[\mathsf{m}(\tilde{x}@).P] \qquad \text{(dep.act.bound)}$$

Local transitions: (x' and $\tilde{y}'$ are fresh)

$$L, u@v \vdash u_v{:}[\mathsf{m}\tilde{x}.P],\ v{:}[\,] \longrightarrow^0 u_v{:}[\,],\ v{:}[\mathsf{m}\tilde{x}.P] \qquad \text{(migrate.at)}$$
$$L \vdash u{:}[\mathsf{out}\tilde{x}.P \mid \mathsf{in}\tilde{y}.Q] \longrightarrow^0 u{:}[\tilde{x}{=}\tilde{y} \mid P \mid Q] \qquad \text{(int)}$$
$$L, \tilde{y}'@\tilde{x} \vdash u{:}[\mathsf{out}\tilde{x}.P \mid \mathsf{in}(\tilde{y}@).Q] \longrightarrow^0 (\tilde{y}')\ u{:}[\tilde{x}{=}\tilde{y}' \mid P \mid Q\{\tilde{y}'/\tilde{y}\}],\ \tilde{y}'{:}[\,] \qquad \text{(int.bound)}$$

$$L, u@v \vdash u{:}[v{=}y],\ v_p{:}[\,] \longrightarrow^0 u{:}[\,],\ v_y{:}[y{=}p],\ \textit{if } v < y \qquad \text{(dep.fu.at)}$$
$$L, u@v \vdash u{:}[v^{\mathsf{m}}\tilde{x}.P],\ v{:}[\,] \longrightarrow^0 u{:}[\,],\ v{:}[\mathsf{m}\tilde{x}.P] \qquad \text{(dep.act.at)}$$
$$L, u@v \vdash u{:}[v^{\mathsf{m}}(\tilde{x}@).P],\ v{:}[\,] \longrightarrow^0 u{:}[\,],\ v{:}[\mathsf{m}(\tilde{x}@).P] \qquad \text{(dep.actb.at)}$$
$$L \vdash u{:}[(x)P\,] \longrightarrow^0 (x')\ u_p{:}[P\{x'/x\}],\ x'{:}[\,] \qquad \text{(dep.new)}$$
$$L, x'@y \vdash u{:}[(x@y)P] \longrightarrow^0 (x')\ u{:}[P\{x'/x\}],\ x'{:}[\,] \qquad \text{(dep.new.at)}$$

Closure of the above rules under contexts:

$$\frac{L \vdash C \longrightarrow^i (\tilde{x})\ C' \qquad \operatorname{chan} C_2 \cap \operatorname{chan} C' = \emptyset}{L \vdash (\tilde{y})\ C, C_2 \longrightarrow^i (\tilde{x}\tilde{y})\ C', C_2}$$

Sequencing of the above rules:

$$\frac{L \vdash M \longrightarrow^i M' \qquad L \vdash M' \longrightarrow^j M''}{L \vdash M \longrightarrow^{i+j} M''}$$

We now explain the transitions. As described in Section 5.5 (page 88), *migrate.at* can be accomplished without messages and in constant time by using linked lists with tail-pointers; and *dep.new.at* and *dep.new* can be accomplished without messages by using lazy channel creation. The transitions *dep.fu* and *dep.act* have optimised local versions *dep.fu.at* and *dep.act.at*, for the case where the terms are being deployed to a co-located machine. And as for the interaction transitions, they are always performed locally.

We illustrate with the example

$$(x@y)(\overline{u}x \mid uy \mid \overline{x}) \longrightarrow (x@y)(x{=}y \mid \overline{x}) \equiv (x@y)(x{=}y \mid \overline{y}).$$

In the fusion machine calculus, this is

$$\begin{aligned} x@y \vdash u{:}[\mathsf{out}x \mid \mathsf{in}y],\ x{:}[\mathsf{out}],\ y{:}[\,] \\ \longrightarrow^0 \quad u{:}[x{=}y],\ x{:}[\mathsf{out}],\ y{:}[\,] \\ \longrightarrow^1 \quad u{:}[\,],\ x_y{:}[\mathsf{out}],\ y{:}[\,] \\ \longrightarrow^0 \quad u{:}[\,],\ x_y{:}[\,]\,y{:}[\mathsf{out}] \end{aligned}$$

which amounts to a $\longrightarrow^1$ transition from the initial machine to the final machine. This can be deduced from the transition rules as follows.

$$\text{from (int)} \quad x@y \vdash u{:}[\mathsf{out}x \mid \mathsf{in}y] \longrightarrow^{0} u{:}[x{=}y] \tag{20}$$

$$\text{(20,context)} \quad x@y \vdash u{:}[\mathsf{out}x \mid \mathsf{in}y],\ x_0{:}[\mathsf{out}],\ y{:}[\,] \longrightarrow^{0} u{:}[x{=}y],\ x_0{:}[\mathsf{out}],\ y{:}[\,] \tag{21}$$

$$\text{(dep.fu)} \quad x@y \vdash u{:}[x{=}y],\ x_0{:}[\,] \longrightarrow^{1} u{:}[\,],\ y_y{:}[\,] \tag{22}$$

$$\text{(22,context)} \quad x@y \vdash u{:}[x{=}y],\ x_0{:}[\mathsf{out}],\ y{:}[\,] \longrightarrow^{1} u{:}[\,],\ x_y{:}[\mathsf{out}],\ y{:}[\,] \tag{23}$$

$$\text{(migrate.at)} \quad x@y \vdash x_y{:}[\mathsf{out}],\ y{:}[\,] \longrightarrow^{1} x_y{:}[\,],\ y{:}[\mathsf{out}] \tag{24}$$

$$\text{(24,context)} \quad x@y \vdash u{:}[\,],\ x_y{:}[\mathsf{out}],\ y{:}[\,] \longrightarrow^{1} u{:}[\,],\ x_y{:}[\,],\ y{:}[\mathsf{out}] \tag{25}$$

$$\text{(21,23,sequence)} \quad x@y \vdash u{:}[\mathsf{out}x \mid \mathsf{in}y],\ x_0{:}[\mathsf{out}],\ y{:}[\,] \longrightarrow^{1} u{:}[\,],\ x_y{:}[\mathsf{out}],\ y{:}[\,] \tag{26}$$

$$\text{(26,25,sequence)} \quad x@y \vdash u{:}[\mathsf{out}x \mid \mathsf{in}y],\ x_0{:}[\mathsf{out}],\ y{:}[\,] \longrightarrow^{1} u{:}[\,],\ x_y{:}[\,],\ y{:}[\mathsf{out}] \tag{27}$$

The costed transition relation will be used in Section 6.10 to judge efficiency. In particular, we will show that *flattening*, introduced in the following section, is efficient.

6.8 Flattening

In Chapter 1 we introduced the idea of *fragmenting* a program—dividing it up into parts, and pre-deploying those parts. This section shows that fragmentation can be encoded purely within the explicit fusion calculus. In fact, the encoding we introduce is not just a fragmentation but a *flattening*—that is, a fragmentation down to the level of individual input and output atoms.

As explained in Section 1.4, the motivation of flattening is to reduce the total *volume* of all messages. We will also show 6.10 that it does not unnecessarily increase the total *number* of messages.

In writing the encoding, we will use co-location commands. These commands were introduced in Chapter 5 (page 88) and defined formally in the previous section (Definition 92, page 111). We use them because the encoding involves a number of extra local channels: it is only through making these channels local to the place where they will eventually be used that we can avoid extra inter-location messages.

Our encoding is a flattening down to the level of individual input and output commands. That is to say, every command that was syntactically guarded in the original is semantically guarded in the result. For instance, we will relate the term $\overline{u}.(\overline{v} \mid v)$ to

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

In this example, the commands $\overline{v}'$ and v'' will necessarily remain idle until after $\overline{u}$ has reacted: thus, we have succeeded in turning a syntactic guard into a semantic guard. Moreover, since v' and v'' are co-located with v, it will take no inter-location messages to rename the guarded body.

My flattening 'flat' is given below. Laneve and Victor [34] have also given a different flattening 'cflat'. This will be analysed in detail in Section 6.10. It was a key inspiration to my own work. What follows is a list of the key differences.

1. My encoding preserves strong reduction-closed congruence, so that $P \sim_b Q$ implies $\text{flat}\, P \sim_b \text{flat}\, Q$. But the encoding of Laneve and Victor only preserves weak congruence, since it involves extra computational steps: $P \approx Q$ implies $\text{cflat}\, P \approx \text{cflat}\, Q$.

2. My encoding is also itself a congruence: $P \sim_b \text{flat}\, P$. This means that any sub-program can be replaced by its flattened form, within any larger context. By contrast, the encoding of Laneve and Victor is not a congruence, and must instead be applied to an entire program.

3. My encoding permits efficient distributed computation. By contrast, a program $\text{cflat}\, P$ must either be executed locally, or must cost extra inter-location messages.

4. My encoding uses a prefix operator upon fusions. The encoding of Laneve and Victor is more elegant, since it dispenses completely with the prefix operator.

5. My encoding is idempotent: $\text{flat}\,\text{flat}\, P \equiv \text{flat}\, P$. The encoding of Laneve and Victor is not.

Technically, we will relate terms P to triples of the form

$$(\tilde{x}_l,\ \phi,\ P').$$

This triple should be understood as the term $(\tilde{x}_l)(\phi \mid P')$, in which P' contains no nested actions and has $\text{Eq}(P') = \mathbf{I}$, and the alpha-renamable locators $\tilde{x}_l$ correspond to the top level actions in P in a sense to be defined (Lemma 96). We will show that the term $(\tilde{x}_l)(\phi \mid P')$ is bisimilar to P. We now give the definition of flattening, and follow it with a worked example.

Definition 95 (Flattening) *The function $[\![\cdot]\!]$ from terms in the explicit fusion calculus to triples $(\tilde{x}_l, \phi, P')$, and the function $\text{flat}\,\cdot$ from terms in the calculus to flat terms in the calculus, are as follows. In the parallel rule, suppose by alpha-renaming that $\tilde{x}'$ does not clash with $\{\tilde{y}'\} \cup \text{fn}(\psi|Q')$ and that $\tilde{y}'$ does not clash with $\{\tilde{x}'\} \cup \text{fn}(\phi|P')$. In the prefix rule, let u' be fresh and suppose that $\{\tilde{x}\} \cap \{\tilde{z}\} = \emptyset$.*

$$\begin{aligned}
[\![\boldsymbol{0}]\!] &= (\emptyset,\ \emptyset,\ \boldsymbol{0}) \\
[\![x{=}y]\!] &= (\emptyset,\ x{=}y,\ \boldsymbol{0}) \\
[\![(z)P]\!] &= (z\tilde{x}_l,\ \phi,\ P') \text{ where } [\![P]\!]{=}(\tilde{x}_l, \phi, P') \text{ and } z \notin \tilde{x}_l \\
[\![u^{\mathsf{m}}\tilde{z}.P]\!] &= \big(u'@u,\ u{=}u',\ (\tilde{x}_l)(u'^{\mathsf{m}}\tilde{z}.\phi \mid P')\big) \text{ where } [\![P]\!]{=}(\tilde{x}_l, \phi, P') \\
[\![P \mid Q]\!] &= (\tilde{x}'_l\tilde{y}'_l,\ \phi|\psi,\ P'|Q') \text{ where } [\![P]\!]{=}(\tilde{x}_l, \phi, P') \text{ and } [\![Q]\!]{=}(\tilde{y}_l, \psi, Q') \\
\\
\text{flat}\, P &= (\tilde{x}_l)(\phi \mid P') \text{ where } [\![P]\!]{=}(\tilde{x}_l, \phi, P')
\end{aligned}$$

We give a worked example to illustrate the definition. Our example is the

term $\overline{u}.(\overline{v} \mid v)$:

$$\begin{aligned}
[\![\overline{v}]\!] &= (v'@v,\ v{=}v',\ \overline{v}') \\
[\![v]\!] &= (v''@v,\ v{=}v'',\ v'') \\
[\![\overline{v}|v]\!] &= (v'@v\, v''@v,\ v{=}v'{=}v'',\ \overline{v}'|v'') \\
[\![\overline{u}.(\overline{v}|v)]\!] &= (u'@u,\ u{=}u',\ (v'@v\, v''@v)(\overline{u}'.(v{=}v'{=}v'')|\overline{v}'|v''))
\end{aligned}$$

$$\begin{aligned}
\operatorname{flat} \overline{u}.(\overline{v}|v) &= (u'@u)\big(u{=}u' \mid (v'@v\, v''@v)(\overline{u}'.(v{=}v'{=}v'')|\overline{v}'|v'')\big) \\
&\equiv (v'@v\, v''@v)(\overline{u}.(v{=}v'{=}v'') \mid \overline{v}' \mid v'')
\end{aligned}$$

Observe, incidentally, the translation is idempotent up to structural congruence. Therefore, since it is not the identity, the translation is not injective. Appealingly, the translation turns out mainly compositional (except for the prefix operator, obviously):

$$\begin{aligned}
\operatorname{flat} \mathbf{0} &= \mathbf{0} \\
\operatorname{flat} x{=}y &= x{=}y \\
\operatorname{flat}(x)P &= (x) \operatorname{flat} P \\
\operatorname{flat} P|Q &= \operatorname{flat} P \mid \operatorname{flat} Q
\end{aligned}$$

There are two invariants for a triple $(\tilde{x}_l, \phi, P')$. The first is that all possible input and output commands in P' are over names in $\tilde{x}_l$. The second is that they are all over distinct names.

Lemma 96 *Let* $[\![P]\!] = (\tilde{x}_l, \phi, P')$. *Then*

1. *if* $P' \xrightarrow{\mu}$ *then* $\mu \in \{\tilde{x}_l\}$;
2. P' *has no* $\xrightarrow{\tau}$ *transitions.*

Proof. Note that $\mathrm{Eq}(P') = \mathbf{I}$, so any transitions undergone by P' have not been renamed by fusions. The proof follows with a simple induction on the structure of P. □

6.9 Correctness of flattening

We now work towards a proof that the flattening is correct: it is a barbed congruence. Therefore,

$$P \sim_b \operatorname{flat} P.$$

This also implies that it preserves congruence:

$$P \sim_b Q \ \Leftrightarrow\ \operatorname{flat} P \sim_b \operatorname{flat} Q.$$

We use the efficient bisimulation technique developed in Section 3.5 (page 38).

The following lemma shows that flattening is well behaved, in the sense that it preserves structural congruence: $P \equiv Q$ implies $\operatorname{flat} P \equiv \operatorname{flat} Q$. Note that the reverse implication is not true, because the translation is not injective. For instance, $\overline{u}.(x)P$ and $(x)(\overline{u}.P)$ are not structurally congruent, but their flattenings are.

Lemma 97 *$P \equiv Q$ implies* $\text{flat}\, P \equiv \text{flat}\, Q$.

Proof. This is a straightforward induction over the derivation of $P \equiv Q$. We present two interesting cases.

1. The scope law $(u)(P \mid Q) \equiv (u)P \mid Q$ if $u \notin \text{fn}\, Q$. Assume that $[\![P]\!] = (\tilde{x}_l, \phi, P')$ and $[\![Q]\!] = (\tilde{y}_l, \psi, Q')$. Through alpha renaming, assume $\tilde{x}_l$ and $\tilde{y}_l$ are fresh. Flattening the two sides of the law, we get

$$\begin{aligned} \text{flat}(u)(P \mid Q) &\equiv (u\tilde{x}_l\tilde{y}_l)(\phi|\psi|P'|Q') \\ \text{flat}(u)P \mid Q &\equiv (u\tilde{x}_l)(\phi|P') \mid (\tilde{y}_l)(\psi|Q') \end{aligned}$$

   But by assumption $u \notin \text{fn}\, Q$ we can deduce $u \notin \text{fn}(\tilde{y}_l)(\psi|Q')$; hence the result.

2. The congruence law $P \equiv Q$ implies $P|R \equiv Q|R$. Suppose that $[\![P]\!] = (\tilde{x}_l, \phi, P')$, $[\![Q]\!] = (\tilde{y}_l, \psi, Q')$ and $[\![R]\!] = (\tilde{z}_l, \theta, R')$. Flattening the two sides of the law, we get

$$\begin{aligned} \text{flat}\, P|R &\equiv (\tilde{x}_l\tilde{z}_l)(\phi|\theta|P'|R') \\ \text{flat}\, Q|R &\equiv (\tilde{y}_l\tilde{z}_l)(\psi|\theta|Q'|R') \end{aligned}$$

   We can alpha-rename so that $\tilde{x}_l$, $\tilde{y}_l$ and $\tilde{z}_l$ are fresh. Therefore the two equivalences can be rewritten as

$$\begin{aligned} \text{flat}\, P|R &\equiv (\tilde{x}_l)(\phi|P') \mid (\tilde{z}_l)(\theta|R') \\ \text{flat}\, Q|R &\equiv (\tilde{y}_l)(\psi|Q') \mid (\tilde{z}_l)(\theta|R') \end{aligned}$$

   The induction hypothesis is that $\text{flat}\, P \equiv \text{flat}\, Q$; hence the result. □

Lemma 98 $\text{Eq}(P) = \text{Eq}(\text{flat}\, P)$.

Lemma 99

1. *$P \xrightarrow{\mu} I : P'$ implies* $\text{flat}\, P \xrightarrow{\mu} I : \text{flat}\, P'$.
2. *$\text{flat}\, P \xrightarrow{\mu} I : Q'$ implies* $\exists P' : P \xrightarrow{\mu} I : P',\ Q' \equiv \text{flat}\, P'$.

Proof. Recall that μ ranges over $\mathcal{N} \cup \overline{\mathcal{N}}$. For the first part, we do an induction over the derivation of $P \xrightarrow{\mu} I : P'$. From Definition 19 (page 38) there are three cases:

1. $\mu\tilde{z}.P \xrightarrow{\mu} \tilde{z} : P$. Let $(\tilde{x}_l, \phi, P') = [\![P]\!]$. Flattening both sides we get $(\tilde{x}_l)(\mu\tilde{z}.\phi \mid P') \xrightarrow{\mu} \tilde{z} : (\tilde{x}_l)(\phi \mid P')$. And since $\tilde{x}_l$ was chosen fresh, it does not bind $\tilde{z}$, and the transition is valid.
2. For closure under structural congruence use Lemma 97.
3. For closure under restriction and parallel composition, use the fact that flat is compositional with respect to restriction and parallel composition.

For the second part, we do an induction over the derivation of $\text{flat}\, P \xrightarrow{\mu}$. By Proposition 25, this amounts to an induction over the structure of $\text{flat}\, P$. And this amounts to an induction over the structure of P. The terms $\text{flat}\, \mathbf{0}$ and $\text{flat}\, x{=}y$ have no labelled transitions. This leaves three cases.

1. Given a transition $\xrightarrow{\mu}$ undergone by $\text{flat}\,\mu\tilde{z}.P$, we must show that $\mu\tilde{z}.P$ undergoes the same transition. Let $(\tilde{x}_l, \phi, P') = [\![P]\!]$. Then $\text{flat}\,\mu\tilde{z}.P \equiv (\tilde{x}_l)(\mu\tilde{z}.\phi \mid P')$. By Proposition 25, any transition must come from $\mu\tilde{z}.\phi$ or P'. But by Lemma 96, the transitions in P' are only on names in $\tilde{x}$. Therefore, the term can only make a $\xrightarrow{\mu}$ transition to $\tilde{z} : (\tilde{x}_l)(\phi \mid P')$. This is just the flattened form of $\mu\tilde{z}.P \xrightarrow{\mu} \tilde{z} : P$.

2. Given a transition $\xrightarrow{\mu}$ undergone by $\text{flat}(z)P$. This must come from $\text{flat}\,P$ undergoing the same transition with $z \notin \mu$. By the induction hypothesis, so does P; therefore, so does $(z)P$.

3. Given a transition $\xrightarrow{\mu}$ undergone by $\text{flat}\,P|Q$. Therefore, it is also undergone by $\text{flat}\,P \mid \text{flat}\,Q$. By Proposition 25, this comes either from $\text{flat}\,P \xrightarrow{\mu'}$ or $\text{flat}\,Q \xrightarrow{\mu'}$, with $(\text{flat}\,P|\,\text{flat}\,Q) \vdash \mu' = \mu$. By the induction hypothesis, either $P \xrightarrow{\mu'}$ or $Q \xrightarrow{\mu'}$. By Lemma 98, $P|Q \vdash \mu = \mu'$. Therefore $P \mid Q$ undergoes the same transition. □

Lemma 100

1. $P \xrightarrow{?u=v} P'$ *implies* $u{=}v|\,\text{flat}\,P \xrightarrow{\tau} u{=}v|\,\text{flat}\,P'$

2. $\text{flat}\,P \xrightarrow{?u=v} Q'$ *implies* $\exists P' : u{=}v|P \xrightarrow{\tau} u{=}v|P',\ Q' \equiv \text{flat}\,P'$

Proof. For the first part, the proof is by induction on the derivation of $P \xrightarrow{?u=v} P'$. From Definition 19 (page 38) there are four cases.

1. $\overline{u}\tilde{z}.P \mid v\tilde{w}.Q \xrightarrow{?u=v} \tilde{z}{=}\tilde{w} \mid P \mid Q$. Let $(\tilde{x}_l, \phi, P') = [\![P]\!]$ and $(\tilde{y}_l, \psi, Q') = [\![Q]\!]$. Flattening both sides we get $(\tilde{x}_l\tilde{y}_l)(\overline{u}\tilde{z}.\phi \mid P' \mid v\tilde{w}.\psi \mid Q') \xrightarrow{?u=v} (\tilde{x}_l\tilde{y}_l)(\tilde{z}{=}\tilde{w} \mid \phi \mid \psi \mid P' \mid Q')$. Since $\tilde{x}_l$ and $\tilde{y}_l$ were chosen fresh, they do not bind u or v, and the transition is valid.

2. For the other closure properties, the proof is the same as in the previous lemma.

For the second part, we do an induction on the structure of P. The cases $\mathbf{0}$, $x{=}y$ and $\mu\tilde{z}.P$ undergo no $\xrightarrow{?u=v}$ transitions. Therefore we need only consider restriction and parallel composition.

1. Suppose $\text{flat}(z)P$ undergoes a $\xrightarrow{?u=v}$ transition. By Proposition 25 and the compositionality of the flattening operator, the transition is $(z)\,\text{flat}\,P \xrightarrow{?u=v} (z)Q'$, with $z \notin \{u, v\}$ and $\text{flat}\,P \xrightarrow{?u=v} Q'$. By the induction hypothesis there is a P' such that $Q' \equiv \text{flat}\,P'$ and $u{=}v|P \xrightarrow{\tau} u{=}v|P'$. Hence $u{=}v|(z)P \xrightarrow{\tau} u{=}v|(z)P'$ with $(z)Q' \equiv \text{flat}(z)P'$ as required.

2. Suppose $\text{flat}\,P|Q$ undergoes a $\xrightarrow{?u=v}$ transition coming from P alone. (The case for Q alone is similar). Then $\text{flat}\,P \mid \text{flat}\,Q$ undergoes the same transition: namely, $\text{flat}\,P \mid \text{flat}\,Q \xrightarrow{?u=v} \text{flat}\,P' \mid \text{flat}\,Q$. (We used the induction hypothesis to tell that the reaction of $\text{flat}\,P$ alone led to something in the image of flat.) By Proposition 25, this must come from

flat $P \xrightarrow{?x=y}$ flat P', with flat $P \mid$ flat $Q \vdash ?x=y = ?u=v$. By the induction hypothesis, $x=y|P \xrightarrow{\tau} x=y|P'$. Hence $x=y|P|Q \xrightarrow{\tau} x=y|P'|Q$. By Lemmas 98 and 28, $u=v|P|Q \xrightarrow{\tau} u=v|P'|Q$ as desired.

3. Suppose flat $P|Q$ undergoes a $\xrightarrow{?u=v}$ transition involving P and Q together. This is largely the same as the previous case. □

Proposition 101 $P \sim_g$ flat P

Proof. From the previous lemmas, the relation $\mathcal{S} = \{(P, Q) : Q \equiv \text{flat}\, P\}$ is an efficient bisimulation—in other words, flat itself is an efficient bisimulation. Hence, from Proposition 32, it is a ground bisimulation. □

We only defined flattening on terms without replication. Indeed, there seems little point in defining it on terms with replication: the aim of guarded replication $!\overline{u}x.P$ is to *prevent* P from being deployed until it is needed, while the aim of flattening flat P is to *allow* P to be deployed before it is needed.

Even so, because of the congruence result, any P and flat P remain congruent even in contexts which include replication. For instance, given a program $!\mu.P$, we can flatten just the inside to $!\mu.(\text{flat}\, P)$.

Thus, because our flattening is a congruence, there is neither any need nor purpose in flattening replication. In contrast, a different form of flattening by Laneve, Parrow and Victor is not a congruence. Therefore, when they show how to flatten replication [33], it is for them a basic necessity.

6.10 Efficiency of flattening

This section is concerned with the efficiency of flattening. In particular, we show that the flattening from the previous section is efficient. This draws on all the preceding results from this chapter: it uses the fusion machine as a model for where the costs are; it uses the costed transition relation to count these costs; and it uses the flattening and its proof of correctness. We also discuss the efficiency of a *catalyst flattening* presented by Laneve and Victor [34].

There are two ways to judge efficiency: by the total number of messages it takes to reach a given state, or by the total volume of traffic. Consider a program of size n. In the introduction (Section 1.4, page 12) we suggested why a program would normally takes n messages, with total size $\frac{1}{2}n^2$. We also suggested that flattening could achieve $2n$ messages with total size $2n$.

I do not yet know how best to measure either the size of a program or the size of a message. Therefore the scope of this section is a little smaller: we will count the number of messages, but not their size. In particular, we show that if a program takes n messages, then its flattening need take no more than $2n$. Although we do not quantify their size, it will be clear that each of these $2n$ messages are small.

We will start with an example. Consider the program

$$P = \overline{u}.(\overline{v} \mid v) \mid u.$$

Running on a fusion machine, $c{:}[P]$ might execute as follows. We have omitted all empty channel-managers for clarity.

$$\begin{array}{lll}
 & & c{:}[\overline{u}.(\overline{v} \mid v) \mid u] \quad (28)\\
\text{Deploy } \overline{u}{:} & \longrightarrow^1 & c{:}[u],\ u{:}[\mathsf{out}.(\overline{v} \mid v)] \quad (29)\\
\text{Deploy } u{:} & \longrightarrow^1 & u{:}[\mathsf{in} \mid \mathsf{out}.(\overline{v} \mid v)] \quad (30)\\
\text{React at } u{:} & \longrightarrow^0 & u{:}[\overline{v} \mid v] \quad (31)\\
\text{Deploy } \overline{v}{:} & \longrightarrow^1 & u{:}[v],\ v{:}[\mathsf{out}] \quad (32)\\
\text{Deploy } v{:} & \longrightarrow^1 & v{:}[\mathsf{in} \mid \mathsf{out}] \quad (33)\\
\text{React at } v{:} & \longrightarrow^0 & \mathbf{0} \quad (34)
\end{array}$$

Flattening the program yields the following:

$$\text{flat}\, P = (u'@u, u''@u)\Big(u{=}u'{=}u'' \mid (v'@v, v''@v)\big(\overline{u}'.(v{=}v'{=}v'') \mid \overline{v}' \mid v''\big) \mid u''\Big)$$

We will show a possible execution of this flattened program, which takes twice as many messages as the original. Our first step is to deploy all the restrictions using the *(dep.new.at)* rule

$$L, x'@y \vdash u{:}[(x@y)P] \longrightarrow^0 (x')\ u{:}[P\{x'/x\}],\ x'{:}[\,] \qquad x' \textit{ fresh.} \quad \text{(dep.new.at)}$$

Writing the entire execution out in full would be confusing. Instead, we focus on the essential parts. In particular, by an abuse of notation, we assume that all restricted names are already unique so that no renaming is needed. We also assume an L which satisfies all the locating restrictions $(u'@u)\ldots$ in the flattened term, and we omit the resulting restrictions (x') in the machine. The result of deploying all restrictions is as follows:

$$c{:}[\text{flat}\, P] \longrightarrow^0 c{:}[u{=}u'{=}u'' \mid \overline{u}'.(v{=}v'{=}v'') \mid \overline{v}' \mid v'' \mid u].$$

Execution now continues as shown below. The numbering to the side indicates which states in this flattened execution trace, correspond to which states in the original execution trace.

$$\begin{array}{lll}
 & & c{:}[u{=}u'{=}u'' \mid \overline{u}'.(v{=}v'{=}v'') \mid \overline{v}' \mid v'' \mid u] \quad (28')\\
\text{Deploy } \overline{u}{:} & \longrightarrow^1 & c{:}[u{=}u'' \mid \overline{u}'.(v{=}v'{=}v'') \mid \overline{v}' \mid v'' \mid u],\ u'_u{:}[]\\
 & \longrightarrow^1 & c{:}[u{=}u'' \mid \overline{v}' \mid v'' \mid u],\ u'_u{:}[\mathsf{out}.(v{=}v'{=}v'')]\\
 & \longrightarrow^0 & c{:}[u{=}u'' \mid \overline{v}' \mid v'' \mid u],\ u{:}[\mathsf{out}.(v{=}v'{=}v'')] \quad (29')\\
\text{Deploy } u{:} & \longrightarrow^1 & c{:}[\overline{v}' \mid v'' \mid u],\ u''_u{:}[],\ u{:}[\mathsf{out}.(v{=}v'{=}v'')]\\
 & \longrightarrow^1 & c{:}[\overline{v}' \mid v''],\ u''_u{:}[\mathsf{in}],\ u{:}[\mathsf{out}.(v{=}v'{=}v'')]\\
 & \longrightarrow^0 & c{:}[\overline{v}' \mid v''],\ u{:}[\mathsf{in} \mid \mathsf{out}.(v{=}v'{=}v'')] \quad (30')\\
\text{React at } u{:} & \longrightarrow^0 & c{:}[\overline{v}' \mid v''],\ u{:}[v{=}v'{=}v''] \quad (31')\\
\text{Deploy } \overline{v}{:} & \longrightarrow^1 & c{:}[\overline{v}' \mid v''],\ u{:}[v{=}v''],\ v'_v{:}[]\\
 & \longrightarrow^1 & c{:}[v''],\ u{:}[v{=}v''],\ v'_v{:}[\mathsf{out}]\\
 & \longrightarrow^0 & c{:}[v''],\ u{:}[v{=}v''],\ v{:}[\mathsf{out}] \quad (32')\\
\text{Deploy } v{:} & \longrightarrow^1 & c{:}[v''],\ v''_v{:}[],\ v{:}[\mathsf{out}]\\
 & \longrightarrow^1 & v''_v{:}[\mathsf{in}],\ v{:}[\mathsf{out}]\\
 & \longrightarrow^0 & v{:}[\mathsf{in} \mid \mathsf{out}] \quad (33')
\end{array}$$

React at v: $\longrightarrow^0 \quad \mathbf{0}$ (34')

The essential difference between the original and this flattened version is in the deployment of actions. In the original, it took just one transition, costing one potentially large message. In this flattened version it takes three transitions:

1. Deploy a fusion $u = u'$. This costs one small (fixed-size) message.
2. Deploy an action $\overline{u}'.\phi$. This costs one message whose size is only the size of the fusion ϕ: i.e. the number of parallel actions prefixed by $\overline{u}$.
3. Migrate the action from u' to u. This costs nothing, since u' is co-located with u.

We have shown only one possible execution trace, in which the actions $\overline{v}' \mid v''$ are deployed after the fusions $v{=}v'{=}v''$. Another possible trace would have these actions pre-deployed in advance. Now consider a different program in which the actions are pre-deployed but then $u{:}[\mathsf{out}.(\overline{v} \mid v)]$ never reacts for some reason: in this case, the pre-deployment would have been in vain, costing needless messages. On the other hand, since the messages are asynchronous, they would not be a bottleneck for any other part of the computation. Costing these issues is subtle. We will sidestep the issues by proving only an efficiency *simulation*, in which the flattened program is able to match the costs of the original: not an efficiency *bisimulation*, where it is required to match.

Note the possibility where u and v are co-located. In this case, it would cost no messages to deploy the actions $\overline{v} \mid v$ in the original:

$$u{:}[\overline{v} \mid v] \quad \longrightarrow^0 \quad u{:}[v],\ v{:}[\mathsf{out}] \quad \longrightarrow^0 \quad v{:}[\mathsf{in} \mid \mathsf{out}]$$

By contrast, the flattened version would still cost messages to deploy the actions:

$$\begin{array}{lll} c{:}[\overline{v}' \mid v''],\ u{:}[v{=}v'{=}v''] & \longrightarrow^0 & c{:}[\overline{v}' \mid v''],\ u{:}[v{=}v''],\ v'_v{:}[] \\ & \longrightarrow^1 & c{:}[v''],\ u{:}[v{=}v''],\ v'_v{:}[\mathsf{out}] \\ & \longrightarrow^0 & c{:}[v''],\ u{:}[v{=}v''],\ v{:}[\mathsf{out}] \\ & \longrightarrow^0 & c{:}[v''],\ v''_v{:}[],\ v{:}[\mathsf{out}] \\ & \longrightarrow^1 & v''_v{:}[\mathsf{in}],\ v{:}[\mathsf{out}] \\ & \longrightarrow^0 & v{:}[\mathsf{in} \mid \mathsf{out}] \end{array}$$

This raises an apparent discrepancy, of the flattened version costing more than twice the original. It is only an apparent discrepancy, however, which arises from our simplification of counting number of messages rather than size. If we had counted size as well, we would have included the original's initial cost to transport $\overline{v}' \mid v''$ from c to u. To count size as well is beyond the scope of this chapter. Instead, to avoid the discrepancy, we will assume that no names in the original program P are co-located.

Proposition 102 (Efficiency) *If $L \vdash c{:}[P] \longrightarrow^i M'$ where no names in P are co-located, then there exists a machine N' such that $L \vdash c{:}[\mathrm{flat}\, P] \longrightarrow^j N'$ such that $j \leq 2i$ and $M' \dot{\sim}_b N'$.*

Proof. We will define a relation $\mathcal{S}$ on machines such that $c{:}[P]\ \mathcal{S}\ c{:}[\mathrm{flat}\, P]$. We will establish that $\mathcal{S}$ is closed with respect to costed transitions in M, and

satisfies $M \dot{\sim}_b N$. This constitutes a proof of the proposition. We might call $\mathcal{S}$ an *efficiency simulation*.

As in the example given above, $c{:}[\text{flat } P]$ will leave in c all actions, right until they are about to be used. By contrast, $c{:}[P]$ moves actions about according to the name that guards them. We will quotient this difference out, as follows. Define $\equiv_{\text{ni}}$ to be the smallest congruence containing $\equiv$ such that

$$x{:}[P \mid B_1],\ y{:}[B_2] \quad \equiv_{\text{ni}} \quad x{:}[B_1],\ y{:}[P \mid B_2].$$

The subscript *ni* stands for the fact that it is *not important* where in the machine a body P should be. We adopt a notation which subsumes this quotient: write all terms P in parallel with channel-managers, rather than inside them. For instance, both sides of the above equation are represented by $x{:}[B_1], P, y{:}[B_2]$. We will also assume but not write down the empty channel-managers.

This quotient is reminiscent of the uniprocessor machine of Pierce and Turner (described in Section 1.3 page 10), in which all bodies P are placed in a central location. However, we are using the quotient merely as a proof technique; the uniprocessor machine uses it as an implementation technique.

We define flattening on machines. The function flat M is obtained by piecewise application of flat to the body of each channel-manager in M as follows:

$$\begin{aligned} \text{flat } B_1|B_2 &= \text{flat } B_1 \mid \text{flat } B_2 \\ \text{flat } P &= (\tilde{x}_l)(\phi \mid P') \text{ where } [\![P]\!]{=}(\tilde{x}_l, \phi, P') \\ \text{flat } \mathsf{m}\tilde{z}.P &= (\tilde{x}_l)(\mathsf{m}\tilde{z}.\phi \mid P') \text{ where } [\![P]\!]{=}(\tilde{x}_l, \phi, P') \end{aligned}$$

Define the relation $\mathcal{S}$ such that $M\ \mathcal{S}\ N$ if and only if $N \equiv_{\text{ni}} \text{flat } M$. One issue is that the translation cflat P will introduce many local names. Because we have not defined garbage collection on the machine, there will be many local names left behind after the execution of $c{:}[\text{flat } P]$, which are not produced by $c{:}[P]$. Strictly speaking, the relation $\mathcal{S}$ should retain all these names. But because this introduces a lot of book-keeping, we will instead just assume garbage collection in the machine.

Clearly, $c{:}[P]\ \mathcal{S}\ c{:}[\text{flat } P]$.

We now prove that $\mathcal{S}$ is closed under transitions. In particular, if $M\ \mathcal{S}\ N$ and $M \longrightarrow^i M'$, then there exists an N' such that $N \longrightarrow^j N'$ with $j \leq 2i$, and $M'\ \mathcal{S}\ N'$. We do this by induction on the derivation of $M \longrightarrow^i M'$ (Definition 94, page 111).

1. *(migrate)* Suppose

   $$L \vdash\ u_v{:}[\mathsf{m}\tilde{z}.P], v{:}[] \quad \longrightarrow^1 \quad u_v{:}[], v{:}[\mathsf{m}\tilde{z}.P].$$

   Let $(\tilde{x}_l, \phi, P') = [\![P]\!]$. Then, flattening the left hand side and quotienting by $\equiv_{\text{ni}}$ we get $u_v{:}[\mathsf{m}\tilde{z}.\phi], v{:}[], P'$. This can undergo a $\longrightarrow^1$ transition to $u_v{:}[], v{:}[\mathsf{m}\tilde{z}.\phi], P'$. The result of this transition is just the same as the flattening and quotienting of the right hand side in the original, as desired.

2. *(dep.fu)* Suppose

   $$u{:}[v{=}y], v_p{:}[] \quad \longrightarrow^1 \quad u{:}[], v_y{:}[y{=}p], \quad \text{if } \mathrm{v} \text{¦} \mathrm{y}$$

   Again, flattening and quotienting the left hand side allows for a matching *(dep.fu)* transition.

3. *(dep.act)* Suppose

$$u{:}[v^{\mathsf{m}}\tilde{z}.P], v{:}[] \quad \longrightarrow^1 \quad u{:}[], v{:}[\mathsf{m}\tilde{z}.P].$$

Flattening and quotienting the left hand side we get $(v'@v)(v'{=}v \mid v'^{m}\tilde{z}.\phi \mid P')$. This reacts as follows:

$$\begin{aligned}(v'@v)(v'{=}v \mid v'^{m}\tilde{z}.\phi \mid P') \quad &\longrightarrow^0 \quad (\!(v')\!)\ v'{=}v \mid v'^{m}\tilde{z}.\phi \mid P' \\ &\longrightarrow^1 \quad (\!(v')\!)\ v'_v{:}[], v'^{m}\tilde{z}.\phi, P' \\ &\longrightarrow^1 \quad (\!(v')\!)\ v'_v{:}[\mathsf{m}\tilde{z}.\phi], P' \\ &\longrightarrow^0 \quad (\!(v')\!)\ v'_v{:}[], v{:}[\mathsf{m}\tilde{z}.\phi], P'\end{aligned}$$

Since we have assumed garbage collection, and since the name v' does not appear in the rest of the term, this is just the same as $v{:}[\mathsf{m}\tilde{z}.\phi], P'$. And this is just the flattening and quotienting of the original right hand side. Deploying a bound action is similar.

4. *(int)* Suppose

$$u{:}[\mathsf{out}\tilde{z}_1.P \mid \mathsf{in}\tilde{z}_2.Q] \quad \longrightarrow^0 \quad u{:}[\tilde{z}_1{=}\tilde{z}_2 \mid P \mid Q].$$

Let $[\![P]\!] = (\tilde{x}_l, \phi, P')$ as before, and also $[\![Q]\!] = (\tilde{y}_l, \psi, Q')$. Flattening and quotienting the left hand side, we get

$$P', Q', u{:}[\mathsf{out}\tilde{z}_1.\phi \mid \mathsf{in}\tilde{z}_2.\psi] \quad \longrightarrow^0 \quad P', Q', \tilde{z}_1{=}\tilde{z}_2, \phi, \psi.$$

But this is just the same as the flattening and quotienting of the right hand side.

5. The co-location transitions in M are not possible, since we assumed no names in P to be co-located. □

This concludes the induction. Finally, observe that $\mathrm{calc}\,\mathrm{flat}\,M \equiv \mathrm{flat}\,\mathrm{calc}\,M$. Therefore (Proposition 101) for any $M \ \mathcal{S} \ N$, $\mathrm{calc}\,N \ \dot{\sim}_b \ \mathrm{calc}\,M$. Therefore (Corollary 85) $M \ \dot{\sim}_b \ N$ as desired.

We now consider the efficiency of another flattening given by Laneve and Victor. Their flattening was a key influence on mine. In fact, they were motivated not by practical concerns of efficiency, but by theoretical concerns of elegance. Its elegance lies in the fact that it manages to avoid all use of the prefix operator; by contrast, my flattening uses a prefix operator on fusions. However, the elegance comes at the cost of preventing pre-deployment.

Their flattening, cflat, uses *catalyst agents* $U_y = (z)y\,zzy$ which have the effect of fusing two names that they receive. For instance, the following example uses the catalyst U_y to fuse $u{=}v$.

$$\begin{aligned}U_y \mid (w)(\overline{y}\,uvw \mid U_w) \quad &\equiv \quad (z)y\,zzy \mid (w)(\overline{y}\,uvw \mid U_w) \\ &\longrightarrow \ u{=}v \mid U_y.\end{aligned}$$

There is also a subsidiary flattening cflat_y, which yields a term that is currently blocked but which can be unblocked by a catalyst at y. We now define both cflat and cflat_y. The definition is a little hard to read, so we follow it immediately with a worked example.

Definition 103 (Catalyst flattening) *Assume w, y, y', y'' not free in P. Let $U_y = (z)y\,zzy$. Then*

$$\mathrm{cflat}\,P \;=\; (y)(\mathrm{cflat}_y\,P \mid U_y)$$

where the subsidiary translation cflat_y*, parameterised by y, is as follows:*

$$\begin{aligned}
\mathrm{cflat}_y\ \overline{u}\tilde{x}.P &= (wy'y'')\big(\overline{w}\tilde{x}yy'' \mid \overline{y}uwy' \mid U_{y'} \mid \mathrm{cflat}_{y''}\,P\big)\\
\mathrm{cflat}_y\ u\tilde{x}.P &= (wy'y'')\big(w\tilde{x}y''y \mid \overline{y}uwy' \mid U_{y'} \mid \mathrm{cflat}_{y''}\,P\big)\\
\mathrm{cflat}_y\ (x)P &= (x)(\mathrm{cflat}_y\,P)\\
\mathrm{cflat}_y\ P \mid Q &= \mathrm{cflat}_y\,P \mid \mathrm{cflat}_y\,Q
\end{aligned}$$

As an example, we flatten the term $\overline{u}.(\overline{v}x)$. This involves many local names. In general, for $\mathrm{cflat}_y\,\mu.P$ we use three local names: w which will become fused to μ when the action has been unlocked; y' which will become fused to the catalyst y when the action has been unlocked, so as to unlock other parallel actions; and y'' which will become fused to the catalyst y which this action μ is consumed in reaction. We use the subscript 0 for the three names when they are used in $\mathrm{cflat}_y\,\overline{u}$, and 1 for the three names when they are used in $\mathrm{cflat}_{y''_0}\,\overline{v}x$. In fact, the same name is then addressed by both y''_0 (to lock the term guarded by $\overline{u}$), and y_1 (for the name under which $\overline{v}x$ is locked). Writing y_1 for this name, the translation $\mathrm{cflat}\,\overline{u}.(\overline{v}x)$ is

$$(y\,w_0\,y'_0\,y_1\,w_1\,y'_1\,y''_1)\big(U_y \mid U_{y'_0} \mid U_{y'_1} \mid \overline{y}uw_0y'_0 \mid \overline{w}_0yy_1 \mid \overline{y}_0vw_1y'_1 \mid \overline{w}_1xy_1y''_1\big).$$

Immediately, without any conditions, the top-level actions can be unlocked. This happens by reaction $\overline{y}uw_0y'_0$ with the catalyst U_y. This fuses $u{=}w_0$ and $y'_0{=}y$, giving

$$(y\,y_1\,w_1\,y'_1\,y''_1)\big(U_y \mid U_{y'_1} \mid \overline{u}yy_1 \mid \overline{y}_0vw_1y'_1 \mid \overline{w}_1xy_1y''_1\big).$$

Before anything further can happen, the $\overline{u}$ must react with the environment. Once this happens, it will fuse y to y_1, giving

$$(y\,w_1\,y'_1\,y''_1)\big(U_y \mid U_{y'_1} \mid \overline{y}vw_1y'_1 \mid \overline{w}_1xyy''_1\big).$$

The action $\overline{v}x$ is now at top level and can now be unlocked by reaction with the catalyst, just as in the first step. This will fuse $v{=}w_1$ and $y'_1{=}y$, giving

$$(y\,y''_1)\big(U_y \mid \overline{v}xyy''_1\big).$$

Laneve and Victor have shown that 'cflat' preserves congruence: if $P \approx_b Q$ then $\mathrm{cflat}\,P \approx_b \mathrm{cflat}\,Q$. However, it is not itself a congruence. Contrast this to 'flat', which is a congruence: that is to say, any sub-program P can be replaced by $\mathrm{flat}\,P$ within any larger context, but cannot be replaced by $\mathrm{cflat}\,P$. This is because $\mathrm{cflat}\,P$ requires extra names to be transmitted in every input and output action. Note too that cflat only works up to weak bisimulation, while flat also works up to strong bisimulation. This is because cflat uses extra internal reaction steps.

However, in this section we shall investigate not the bisimulation of cflat but its efficiency. To do this we will use a simpler example program $P = \overline{u} \mid u$. We

will first show the execution trace of this program, assuming no co-location. By drawing a graph of all the messages involved, we will be able to tell how cflat must use co-location to make it more efficient.

When run directly on the machine, the program P executes as follows:

$$c{:}[\overline{u}|u] \longrightarrow^1 c{:}[u],\ u{:}[\mathsf{out}] \longrightarrow^1 u{:}[\mathsf{out} \mid \mathsf{in}] \longrightarrow^0 \mathbf{0}$$

This original takes two messages. Therefore, we expect that the flattened form should take no more than four messages. The flattened form $\mathrm{cflat}\,\overline{u} \mid u$ is

$$\begin{array}{l}(y z_0\ w_1 y_1' y_1'' z_1\ w_2 y_2' y_2'' z_2) \\ \qquad (y\ z_0 z_9 y \mid \overline{y}\,u w_1 y_1' \mid y_1'\ z_1 z_1 y_1' \mid \overline{y}\,u w_2 y_2' \mid \overline{w}_1\ y y_1'' \mid \overline{w}_2\ y_2'' y \mid y_2'\ z_2 z_2 y_2')\end{array}$$

We will trace through the execution of $c{:}[\mathrm{cflat}\,\overline{u}|u]$ on the fusion machine. There are lots of terms, so we introduce some abbreviations. As before we omit empty channel-managers. We write $u \rightsquigarrow v$ to indicate that the channel-manager u has a fusion-pointer to v, and also that $u < v$ in the total order on names.

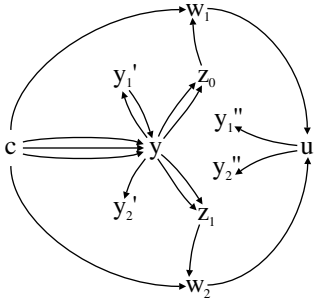
$$\begin{array}{l} A = y\ z_0 z_0 y \mid \overline{y}\,u w_1 y_1' \\ B = y_1'\ z_1 z_1 y_1' \mid \overline{y}\,u w_2 y_2' \\ C = \overline{w}_1\ y y_1'' \mid w_2\ y_2'' y \\ D = y_2'\ z_2 z_2 y_2' \\ E = y_1' \rightsquigarrow y \mid z_0 \rightsquigarrow w_1 \rightsquigarrow u \\ F = y_2' \rightsquigarrow y_1' \rightsquigarrow y \mid z_0 \rightsquigarrow w_1 \rightsquigarrow u \mid z_1 \rightsquigarrow w_2 \rightsquigarrow u \end{array}$$

The total order on names indicated in F above is just one possible order, but the other possible orders make only minor differences to the execution trace.

The first step of executing $c{:}[\mathrm{cflat}\,\overline{u}|u]$ is to deploy all the restrictions, just as before. Execution then continues as follows.

		$c{:}[ABCD]$
Deploy actions A from c:	$\longrightarrow^2$	$c{:}[BCD],\ y{:}[\mathsf{in}\ z_0 z_0 y \mid \mathsf{out}\ u w_1 y_1']$
Interact at y:	$\longrightarrow^0$	$c{:}[BCD],\ y{:}[z_0{=}u \mid z_0{=}w_1 \mid y_1'{=}y]$
Deploy fusions from y:	$\longrightarrow^4$	$c{:}[BCD],\ E$
Deploy actions B from c:	$\longrightarrow^2$	$c{:}[CD],\ y{:}[\mathsf{out}\ u w_2 y_2'],\ y_1'{:}[\mathsf{in}\ z_1 z_1 y_1'],\ E$
Migrate from y_1' to y:	$\longrightarrow^1$	$c{:}[CD],\ y{:}[\mathsf{out}\ u w_2 y_2' \mid \mathsf{in}\ z_1 z_1 y_1'],\ E$
Interact at y:	$\longrightarrow^0$	$c{:}[CD],\ y{:}[z_1{=}u \mid z_1{=}w_2 \mid y_2'{=}y_1'],\ E$
Deploy fusions from y:	$\longrightarrow^4$	$c{:}[CD],\ F$
Deploy actions C from c:	$\longrightarrow^2$	$c{:}[D],\ w_1{:}[\mathsf{out}\ y y_1''],\ w_2{:}[\mathsf{in}\ y y_2''],\ F$
Migrate $w_1 \rightsquigarrow u$, $w_2 \rightsquigarrow u$:	$\longrightarrow^2$	$c{:}[D],\ u{:}[\mathsf{out}\ y y_1'' \mid \mathsf{in}\ y y_2''],\ F$
Interact at u:	$\longrightarrow^0$	$c{:}[D],\ u{:}[y_1''{=}y \mid y_2''{=}y],\ F$
Deploy fusions from u:	$\longrightarrow^2$	$c{:}[D],\ y_1'' \rightsquigarrow y,\ y_2'' \rightsquigarrow y,\ F$

The following graph shows all messages involved in the execution. Each directed edge corresponds to one message.

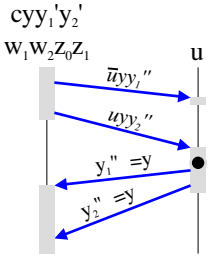


Our task is to figure which names should be co-located by the translation cflat, to have as few inter-location messages as possible. In effect, we must partition the graph so that few edges cross between partitions. But there are two constraints. First, the translation cflat is not at liberty to choose locations for u and c, so we must assume them to be in different partitions. Second, although it is not apparent in the short example program $\overline{u}|u$, in a larger program there will be several subsequent messages between $\{y_1'', y_2''\}$ and y. Therefore, y_1'' and y_2'' should be co-located with y.

There is only one partition which uses no more than four inter-location messages: all names are co-located with c, apart from u. Using this partition, we now present an efficient form of cflat. It is predicated upon the channel c from which the flattened program will be deployed.

$$\begin{aligned}
\text{cflat}_c\, P &= (y@c)(\text{cflat}_{c,y}\, P \mid U_y) \\
\\
\text{cflat}_{c,y}\ \overline{u}\tilde{x}.P &= (w@c, y'@c, y''@c)\big(\overline{w}\tilde{x}yy'' \mid \overline{y}uwy' \mid U_{y'} \mid \text{cflat}_{c,y''}\, P\big) \\
\text{cflat}_{c,y}\ u\tilde{x}.P &= (w@c, y'@c, y''@c)\big(w\tilde{x}y''y \mid \overline{y}uwy' \mid U_{y'} \mid \text{cflat}_{c,y''}\, P\big) \\
\text{cflat}_{c,y}\ (x)P &= (x)(\text{cflat}_{c,y}\, P) \\
\text{cflat}_{c,y}\ P \mid Q &= \text{cflat}_{c,y}\, P \mid \text{cflat}_{c,y}\, Q
\end{aligned}$$

This located catalyst flattening cflat uses a total of $2n$ messages, just like our earlier flattening. However, cflat has a striking property which we illustrate in the following interaction diagram. The diagram shows all the inter-location messages sent during the execution of c:[cflat $\overline{u}|u$]. We have drawn all the channel-managers $cyy_1'y_2''w_1w_2z_0z_1$ in the same vertical column, since they are co-located.



Observe how, through our pursuit of a small message count, we have ended up with a flattening which is not usefully distributed, and which uses handshaking. It is basically the same as the Facile machine (Figures 1 and 2 on page 13). In effect, cflat can either be distributed (by placing the catalysts at different locations), or it can use a small number of messages, but it cannot have both.

By contrast, my translation 'flat' is distributed, uses a small number of messages, and avoids handshaking. It correctly implements the deployment machine (Figure 6 on page 16).

Conclusions. This chapter has developed techniques to reason about distributed efficiency in the pi calculus, and applied them. As far as I know, this has not been done before.

Efficiency can only be judged with respect to some model of execution. We have provided this here by the fusion machine and the co-location assumption L. We then introduced the costed transition relation, counting the number of inter-location messages. We demonstrated how these costed messages could be used: first in an efficiency simulation to prove that one flattening is efficient, and second in a message graph to deduce which names should be co-located in another flattening.

What becomes apparent, however, is that message-counting gives only a partial model of efficiency. We have observed its limitations in two ways. First, since we failed to count message size, we were able to judge flattening's efficiency only in a program that does not already use co-location (Proposition 102). Second, although the catalyst flattening is inefficient through being centralised and serialised, this inefficiency is not reflected in its message count.

Nevertheless, our new techniques of co-location and costed transitions were enough to provide some important results. We showed that there are problems with the catalyst encoding 'cflat'. And we showed that, with our encoding 'flat', the fusion machine really does implement the deployment machine that we had proposed in Chapter 1.

Chapter 7

Conclusions

The word 'fusion' comes from the Latin *fundare*, to pour. Figuratively it describes two metals that have melted and mixed to become one. This is why we call things fused that are identical, indistinguishable, interchangeable.

We have seen several different classes of entities that allow a context to use names interchangeably. In the fusion machine, and through its transitions, fusion-pointers allow names to be used interchangeably. In the calculus, and through structural congruence, the term $x{=}y$ allows names to be used interchangeably. Also in the calculus, and through internal reaction, equators allow names to be used interchangeably. It seems appropriate to define explicit fusions generally: *an explicit fusion is anything which has prolonged existence and which, through some mechanism, allows two names to be used interchangeably.*

A widely studied explicit fusion is Frege's statement over a hundred years ago that the Morning Star is the Evening Star [20]. By this he means that the two names refer to the same object (Venus) even though they imply different modes of presentation (one in the morning, the other in the evening). Much philosophical effort has gone into the observation that the names are interchangeable only in contexts about their objects ('The Morning Star is the second planet from the sun') rather than their mode of presentation ('The Morning Star is visible in the evening').

By contrast, we started from the assumption that explicit fusions allow names to be interchanged in all contexts. One possible mechanism for this interchange is that the names refer to the same object, as for Frege; but it is not the only possibility. Other mechanisms include fusion-pointers, explicit fusion terms $x{=}y$, and equators.

There are three sections in this final chapter. The first section reviews the results of the dissertation. The second section speculates on a wider theory of explicit fusions. And the third section speculates on how the fusion machine might be adopted by working programmers.

7.1 Review

We have introduced the explicit fusion calculus. It provides a new model for synchronous rendezvous: the rendezvous gives rise to an explicit fusion, which then has subsequent substitutive effect. Through this model we are able to

implement distributed synchronous rendezvous without handshaking.

We have developed the bisimulation theory for the explicit fusion calculus. Bisimulation in the explicit fusion calculus turns out to have a number of appealing properties: barbed congruence coincides with ground congruence, which moreover coincides with hyper-equivalence in the fusion calculus. Additionally, for a bisimulation to be a congruence, it turns out to be necessary and sufficient for it to have the same explicit fusions on the inside, and behave the same with any explicit fusions outside. Based on these conditions, we found an efficient characterisation for barbed and ground congruence. As a technical aid in the efficient bisimulation, we introduced the 'ask' fusion labelled transition. A similar transition is also being used in current work by Milner [46] to systematically generate bisimulation congruences for a variety of calculi. If it can be shown that he is using the label in the same way as us, then our results will mean that he is in fact generating the familiar ground congruence.

In addition to these theoretical results, it turns out that the explicit fusion calculus is also practically useful in implementing the pi calculus.

We have introduced the fusion machine. It is a distributed abstract machine for both the pi calculus and the explicit fusion calculus. That is to say: when it is given a term in the explicit fusion calculus, our machine implements it directly. And when given a term in the pi calculus, our machine also implements it directly, without the need first to translate the term into the explicit fusion calculus.

However, there is a compelling reason to translate terms into the explicit fusion calculus before executing them: this allows them to be fragmented. Fragmentation is when we split up a term and pre-deploy the fragments. This in turn allows for a more efficient implementation. (We elaborate on this point below.)

We have shown a way to encode fragmentation purely within the explicit fusion calculus. We have proved that our encoding has a number of appealing properties. Most importantly, it is a congruence. This means that any program in the explicit fusion calculus can be replaced by its flattened form, in any context. It also works up to strong bisimulation: the encoded program takes exactly the same number of reaction steps as the original. It is idempotent. And it does not cost extra messages when implemented in the fusion machine. By contrast, the *catalyst* encoding of Laneve and Victor [34] only works up to weak bisimulation, is not a congruence, and is not idempotent. As for efficiency, it must either cost extra messages, or not be distributed.

As mentioned, the fusion machine implements distributed synchronous rendezvous without handshaking. This is useful, and has not been managed by any previous distributed abstract machines for concurrent calculi. I now consider whether the fusion machine is the *natural and inevitable* distributed implementation of the pi calculus. There are two key questions: whether fragmentation should indeed be used, and whether explicit fusions are the best way to implement this fragmentation.

Fragmentation. We built the fusion machine so that in each deployment message it transports an entire continuation to a channel-manager (through the *(dep.act)* transition given in Section 5.1, and as illustrated in Figure 4 on page 14). We have also demonstrated a way to fragment programs so as to

reduce message volume (through the *flattening* translation given in Section 6.8, and as illustrated in Figure 6 on page 16). The task now is to evaluate whether these techniques are necessary to achieve full efficiency.

To measure efficiency, in terms of counting the number of inter-location messages, we developed the *costed transition relation* (Section 6.7). We also introduced the *costed simulation* technique to compare the cost of two programs. The results of using these techniques are discussed below. As far as I know, such efficiency techniques have not previously been developed for the pi calculus. However, we discovered (Section 6.10) that merely counting the number of messages does not give a complete measure of efficiency: we also need to count the total volume of traffic, and to measure the *latency* of a program—that is, the time between the program receiving some input and replying with the relevant output. Note that, although it is meaningless to count the total cost of an indefinite execution, our approach of counting costs up to some finite stage of execution is valid.

Now if two program fragments wish to rendezvous at a channel, the fusion machine transports them in their entirety to the channel. This takes two messages, one for each. Then rendezvous occurs, and the two program fragments can continue immediately. With this design, the fusion machine requires only two messages per rendezvous. It seems that this design is the only way to achieve such a low message count. For the only alternative would be to leave the continuation behind, as in Facile (Figure 1, page 13); but then it would cost extra messages to tell the continuation to continue.

Section 1.4 conjectured that a program of size n would take n messages to execute, using the design above, with total message volume $\frac{1}{2}n^2$. That is because, in the worst case, the entire program would be transported at every step of its execution. We also conjectured that the same program when flattened would take $2n$ messages with total size $2n$. The first n messages would deploy the (constant-sized) fragments onto the machine, and the second n messages would execute the machine, as above. We proved part of this conjecture using the costed simulation technique: a flattened program does indeed take twice as many messages as the un-flattened version (Section 6.10), and each of these messages is small.

Fusions and forwarders. Section 1.6 proposed two possibilities: we could either implement only those fusions that arise when executing programs in the pi calculus, or we could implement fusions that arise more generally from the explicit fusion calculus.

The discussion in that section revolved around the need for 'conflict resolution' of forwarders. If there exist two forwarders $x \rightsquigarrow y \mid x \rightsquigarrow z$, then two messages sent to x might fail to meet. Hence, the fusion machine resolves the conflict into $x \rightsquigarrow y \mid y \rightsquigarrow z$. This is done with the *(dep.fu)* rule in the fusion machine (Section 5.1).

Now if the machine is limited to only ever running programs in the pi calculus, then this conflict-resolution is never needed. For any reaction $\overline{u}x.P \mid u(y).Q \longrightarrow P \mid Q\{x/y\}$ we can create a forwarder $y \rightsquigarrow x$; the fact that y is bound means that never be any other forwarder $y \rightsquigarrow z$. This argument was made formal in Section 4.5. It introduced *piability*—a structural characterisation for whether a given term in the explicit fusion calculus, is the image under

translation of some term in the pi calculus.

Section 1.6 suggested that there would be no benefit in limiting the fusion machine to purely piable terms. We can now say this more precisely. The general *(dep.fu)* transition is

$$u{:}[x{=}y],\ x_q{:}[\,] \longrightarrow u{:}[\,],\ x_y{:}[y{=}q] \qquad \text{if } x < y.$$

If a term is piable, then q will always be zero, and so the fusion $y{=}q$ in the result will always be dismissed. Therefore, a piable-only fusion machine would still require a *(dep.fu)* transition (albeit a less general one), and it would still execute in exactly the same way as the general rule. I conclude that there is no benefit in limiting the machine to the pi calculus.

Fusions also allow for an appealing programming idiom. Suppose there is a subprogram u which increments a number, and that we wish to increment the number 3. In C++ this would be written

$$\begin{array}{l}\textbf{int}\ u(\textbf{int}\ i)\ \{\textbf{return}\ i+1;\}\\ \ldots\\ \textbf{int}\ y = u(3);\ \mathrm{P};\end{array}$$

We now consider how to implement the program in the pi calculus and the explicit fusion calculus. Integers have not so far been considered in the explicit fusion calculus. I therefore invent the notation $\langle i{+}1\rangle$ just for this discussion, and allow integers as well as names to be transmitted. (Some ramifications are considered in the following section). Now for the pi calculus, and since it lacks fusions, we are forced to invent a *return channel* r along which the answer can be returned:

$$\begin{array}{lcl} & & (r)\overline{u}3r.r(y).P \mid !u(ir).\overline{r}\langle i{+}1\rangle \\ \text{Rendezvous on } u\text{:} & \longrightarrow & (r)(r(y).P \mid \overline{r}4)\ \ldots \\ \text{Rendezvous on } r\text{:} & \longrightarrow & P\{{}^{4}\!/_{y}\}\ \ldots \end{array}$$

This requires two instances of rendezvous, and it blocks the continuation P from executing until the answer has been calculated. But with fusions we can avoid the second rendezvous and the block:

$$\begin{array}{lcl} & & (y)\overline{u}3y.P \mid !u(io).(o{=}\langle i{+}1\rangle) \\ \text{Rendezvous on } u\text{:} & \longrightarrow & (y)(P \mid y{=}4)\ \ldots \\ & \equiv & P\{{}^{4}\!/_{y}\}\ \ldots \end{array}$$

We see that explicit fusions bring benefits with no disadvantages. It makes sense to implement the explicit fusion calculus.

Tree of forwarders. We established (Lemma 73) an invariant about the forwarders in the fusion machine: the forwarders generate a tree, with each edge directed to the root. We also showed that reaction causes two trees to merge, and we showed how this is accomplished by *(dep.fu)* while maintaining the invariant. For this we needed a total order on channel-names, so that the machine's transitions would preserve the invariants.

The efficiency of subsequent transitions will depend on the shape of the tree. For instance, if there is a long chain of forwarders from x to y, then every atom

deployed to x will require a long sequence of migration messages. The shape of the tree will depend on the total order on names, and also on the sequence in which fusions happened to be deployed. I find it hard to predict which tree shapes will appear in practice.

The tree effectively forms a distributed shared state; the state it stores is an equivalence relation on names. This shared state supports query and update operations. To update the shared state, a program can deploy an explicit fusion. (However, the equivalence relation must be monotonically increasing). To query whether two names are related, a program can deploy an output on one and an input on the other. (However, while this will yield a positive answer when the names are related, it will simply do nothing when they are not). The query and update operations can be handled concurrently.

This shared state is relevant to the discussion from Section 1.5. This discussion was about how to implement fragmentation within a calculus. One suggested technique was that each fragment should send its entire environment on to the next fragment. This suggestion was dismissed on the grounds of cost. What the shared state provides is essentially this environment, but without the need to transport it between fragments.

Transitions in calculus and machine. In the explicit fusion calculus, we generally work in equivalence-classes of terms related by structural congruence. The machine has a much smaller structural congruence, and instead accomplishes most of the work through its transitions. This is because the goal of the machine is to model an implementation; and structural congruence is hard to implement.

Therefore, two programs that are related by structural congruence in the calculus are instead related by a sequence of transitions in the machine (up to the tree structure of forwarders).

To bridge this gap between calculus and machine, we divided the machine transitions into two groups, one group $\xrightarrow{\equiv}$ corresponding to structural congruence in the calculus, and the other group $\xrightarrow{\tau}$ corresponding to reactions in the calculus. We then introduced *strong-weak bisimulation*, which is strong with respect to $\xrightarrow{\tau}$ and weak with respect to $\xrightarrow{\equiv}$. Although this mixing of strong and weak is not used elsewhere, it seems a good way to relate a calculus to an implementation.

Mobility. It is sometimes said that the pi calculus models 'mobility without movement': that it models mobility as the making and breaking of links between agents, through communication, but without actually modelling the movement of code. This comment stems from the instinct to add unitary agents into the calculus, to add locations as primitive entities, and to add new commands for the movement of agents between locations.

The comment is not true when we implement the pi calculus with the fusion machine. In the fusion machine, there are no such things as agents, and there are therefore no links between them. The role of communication is to establish forwarders; this is done through the *(dep.fu)* transition. Subsequently, in the presence of a forwarder, a program fragment can move to a new location. When we translate the machine back into the pi calculus (Definition 81), code movement corresponds to alpha-renaming.

One might also say that the explicit fusion calculus models 'dataflow without data flowing': in rendezvous in the pi calculus, it seems that names flow from the sender to the receiver. But in the explicit fusion calculus, all that happens is that the sent names and the received names are fused in the global shared state. There is no directionality in a fusion.

New and restriction. Section 4.5 outlined three roles for restriction in relation to the pi calculus. When converting a piable term in the explicit fusion calculus into a term in the pi calculus, each restriction must play exactly one of the following roles: it can perform the binding in a bound input; it can discharge an explicit fusion; or it can correspond to a normal restriction in the pi calculus. Now the pi calculus uses two operators that bind—input and restriction—while a more parsimonious calculus is possible in which only restriction binds. With the three roles of restriction in piability, we see precisely how the explicit fusion calculus makes do with only restriction.

The fusion machine actually has two roles of restriction itself, which are both mapped to restriction in the explicit fusion calculus. Consider the machine transition $c{:}[(x)P] \longrightarrow (\!|x'|\!)c{:}[P\{x'/x\}]$. Both sides of the transition correspond to the same term $(x)P$ in the calculus, up to alpha renaming. But the machine used one restriction $(x)P$ as a command to create a new name, and it used the other restriction $(\!|x'|\!)$ to record the name that has been created.

Some authors write restriction not as $(x)P$, but as a command `new` x `in` P. This is a good notation for restriction's first role, of creating fresh names. But it is inappropriate for the second role, of recording the *old* names that have already been created.

The operation of the fusion machine is not affected by restriction's second role. Moreover, Section 6.1 established that restriction's second role is irrelevant for the machine's bisimulation congruence. In my opinion, the concept of piability and the design of the fusion machine have helped clarify what we mean by restriction.

7.2 Assumption

This section is motivated by the problem of adding numerical constants into the calculus: *what does it mean to fuse* $1 = 2$*?* The story is told [32] of how the mathematician G.H. Hardy asserted that anything could be proved from this equality. He was challenged by a Mr McTaggart to prove, for example, that he was the Pope. Hardy replied: 'McTaggart and the Pope are two; and two is one; therefore McTaggart and the Pope are one.'

In this section I give more formal grounds for not wanting the fusion. This involves a theory of explicit fusions as *guarded commands*. Perhaps it might be easy just to disallow certain fusions by imposing some arbitrary type system; but the theory in this section will provide a justification and a benchmark for such a type system.

Let us first recall the theory of guarded commands. This theory was first introduced by Dijkstra [13], but the version we use here is a generalisation due to Nelson [49]. Let x range over names, each of which is associated with a number. Let B range over predicates on names, such as $x < 3$. Programs P are

given by

$(x)P$	*non-deterministically choose a value for* x
$P + P$	*non-deterministically choose which program to execute*
$P.P$	*sequential composition*
assert B	*the program aborts if* B *does not hold: disaster*
assume B	*miracle: current execution-trace is only valid if* B *holds*

There are two commands which involve non-deterministic choice. Any given program therefore has a tree of possible execution traces. The 'assume' command culls those sub-branches that fail to satisfy a particular predicate at some stage.

In general, we might implement assumption with back-tracking. However, back-tracking is arduous in an implementation. To avoid it, the language Juno [28, 50] imposes a syntactic constraint: assume can only come directly after the relevant non-deterministic choice. Hence, it is easy to look immediately ahead and make the correct choice. These two programs satisfy Juno's constraint:

$(x)(\text{assume}\, x{=}3.\, P)$	*let* $x = 3$ *in* P
$(\text{assume}\, x{=}3.\, P_1) + (\text{assume}\, x{\neq}3.\, P_2)$	*if* $x = 3$ *then* P_1 *else* P_2

In the first program, the assumption comes after a non-deterministic choice of a value for x. In the second, both assumptions come after a non-deterministic choice as to which branch to take.

I propose that assumptions and guarded commands can be used to interpret the pi calculus, in the same way. Say that the restriction $(x)P$ chooses some channel x non-deterministically. It may choose the same channel as one already existing, or a new channel. The reaction relation means: *we can prove that interaction is always possible, despite the non-determinism.* For instance, in $(x)(\overline{x} \mid y)$, interaction might be possible but only if x happened to be initialised to the same value as y: therefore this term is not part of the reaction relation. On the other hand, in $(x)(\overline{x} \mid x)$, interaction is always possible.

Now the pi calculus has the syntactic constraint that every input name is bound. This is a syntactic constraint in exactly the same sense as Juno—it ensures that every assumption is immediately preceded by a relevant non-deterministic choice. This constraint is similar to the one on piable terms, that every explicit fusion has at least one of its names bound. To illustrate this, let B be the predicate that x and y refer to the same channel. We write assume B instead of the explicit fusion $x{=}y$. Consider the pi calculus program $\overline{u}x \mid u(y).P$ translated into the explicit fusion calculus with assumption:

	$\overline{u}x \mid (y)uy.P$	
$\equiv$	$(y)(\overline{u}x \mid uy.P)$	*interaction is always possible for this term*
$\longrightarrow$	$(y)(\text{assume}\, B \mid P)$	*it is easy to pick a* y *which satisfies* B
$\equiv$	$P\{y/x\}$	B *implies that* x *and* y *refer to the same channel*

We see that an explicit fusion in parallel with a term behaves like an assume command in parallel with a term. Also, note that a match or mismatch operator prefixing a term behaves like an assume command that prefixes the term.

Instead of these constraints, it is possible to use backtracking to get out of any unfulfillable assumption. This is reminiscent of *Distributed Logic Programming*—a language derived from adding synchronous rendezvous to Prolog [14]. In this language, interaction causes the send pattern to be unified with the receive pattern, with possible backtracking in each party. The idea of using guarded commands is also reminiscent of Hoare's *Communicating Sequential Processes* [29], which represents input and output as guarded commands.

We can now answer the initial question regarding what should happen with the explicit fusion $1 = 2$. The answer is that the fusion must never have occurred. If a program reaches that explicit fusion, then its current branch in the execution trace is not a valid one: it must backtrack, and make different decisions at the points of non-determinism. Or, since backtracking is arduous, we might impose some constraints: perhaps constraints on syntax, like Juno and the pi calculus; or perhaps constraints on reaction, like the fusion calculus; or perhaps a type system. Then the explicit fusion assumptions can always be met without backtracking.

7.3 Using the fusion machine

Two questions arise from my work: is it useful to have a concurrent and distributed implementation of the pi calculus or explicit fusion calculus? And is it useful to have *this particular* fusion-machine implementation?

With respect to the second question, note that the pi calculus is not only a programming language but also a notation to describe our intuitions about how distributed programs behave—as a parallel collection of agents which interact. The fusion machine, on the other hand, is a parallel collection of channel-machines, with the programs fragmented between them. What use is it to implement one intuition with a different intuition?

In fact the fusion machine intuition is already used more widely than the agent intuition. A conventional imperative program is a collection of subroutines grouped into modules, with the flow of control jumping between them. And the standard way to implement subroutines in the pi calculus [43] is as replicated input on a channel. Thus, each module is a collection of co-located channels, with continuation messages sent between them—like the fusion machine. Effectively, the fusion machine provides a natural link between conventional programs and the pi calculus.

Also with respect to the second question, consider the matter of *output guards*—that is, output commands followed by some code, such that this code cannot execute until the output command has successfully completed. It is partly in order to support these guards efficiently that the fusion machine is designed as it is, with fragmentation and channel-managers. By contrast, Pict and the join calculus only provide non-guarding input commands: guarding output must be encoded, with a loss of efficiency. Turner [68] reports, from his experience with Pict, that guarding output is in practice rarely used. He also explains that this experience comes from programming in the lambda calculus, and then encoding the lambda calculus into Pict: the encoding itself does not use guarding output. However, I suspect that when we use the pi calculus not to implement other languages but instead to augment them, then it will be more common for programmers to use guarding output. Moreover, they can now use

it safe in the knowledge that the fusion machine implements it efficiently.

With respect to the first question, I believe that a concurrent and distributed implementation of the pi calculus is practically useful. Let us imagine some future programming language which incorporates commands from the pi calculus, and compare it to conventional imperative languages such as C and Java. Every conventional program uses *syntactic guards* (the semicolon) throughout, requiring each command to finish before further commands can be executed. Compilers must try to deduce when this sequencing was intended, and when it was used merely for lack of any other way to compose commands. But perhaps, if synchronous rendezvous and parallel composition were as easy to use as syntactic guards are now, then programmers might use them more frequently—choosing syntactic guards only when dictated by the program's logic. Perhaps, compared to current languages, the resulting programs would be easier to compile and run on multi-processor machines. Certainly they would be easier to write, and easier to understand.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. `http://research.microsoft.com/Users/luca/Papers/ExplicitSub.pdf`

[2] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL '01*, pages 104–115. ACM, ACM Press, 2001. `http://portal.acm.org/citation.cfm?id=360213`

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999. `http://research.microsoft.com/~adg/Publications/spi.ps`

[4] R. M. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed pi-calculus (extended abstract). In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Proceedings of FSTTCS '99*, volume 1738 of *Lecture Notes in Computer Science*, pages 304–315. Springer-Verlag, 1999. `http://www-sop.inria.fr/mimosa/personnel/Gerard.Boudol/rdpi.html`

[5] R. M. Amadio and S. Prasad. Localities and failures. In P. S. Thiagarajan, editor, *Proceedings of FSTTCS '94*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer-Verlag, 1994.

[6] M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. In *Proceedings of EXPRESS '00*, volume 39 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000. To appear. `ftp://ftp.dcs.qmw.ac.uk/lfp/martinb/express00.ps.gz`

[7] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, California, second edition, 1994.

[8] M. Boreale and D. Sangiorgi. Some congruence properties for pi-calculus bisimilarities. *Theoretical Computer Science*, 198(1–2):159–176, 1998. `ftp://ftp-sop.inria.fr/mimosa/personnel/davides/congruence.ps.gz`

[9] L. Cardelli. An implementation model of rendezvous communication. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 449–457. Springer-Verlag, 1984. `http://research.microsoft.com/Users/luca/Papers/Rendezvous.A4.pdf`

[10] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. `http://research.microsoft.com/Users/luca/Papers/MobileAmbients.A4.ps`

[11] L. Cardelli and R. Pike. Squeak: A language for communicating with mice. In B. A. Barsky, editor, *Proceedings of SIGGRAPH '85*, volume 19, pages 199–204. ACM Press, 1985. `http://research.microsoft.com/Users/luca/Papers/Squeak.pdf`

[12] A. de Saint-Exupéry. *Le Petit Prince*. Gallimard, Paris, 1946.

[13] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. `http://portal.acm.org/citation.cfm?id=360975`

[14] A. Eliens. *DLP: A Language for Distributed Logic Programming*. Wiley, Kansas, 1992.

[15] J. Engelfriet and T. Gelsema. Multisets and structural congruence of the pi-calculus with replication. *Theoretical Computer Science*, 211(1–2):311–337, 1999. `http://www.elsevier.com/gej-ng/10/41/16/137/17/24/abstract.html`

[16] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, France, 1998. `http://www.inria.fr/rrrt/tu-0556.html`

[17] C. Fournet, G. Gontheir, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996. `http://research.microsoft.com/~fournet/papers/calculus-of-mobile-agents.ps`

[18] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, ACM Press, 1996. `http://research.microsoft.com/~fournet/papers/reflexive-cham-join-calculus.ps`

[19] C. Fournet, Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Proceedings of IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 348–364. Springer-Verlag, 2000. `http://research.microsoft.com/~fournet/papers/implementation-of-ambients-tcs.pdf`

[20] G. Frege. Über sinn und bedeutung. *Zeitschrift für Philosophie und philosophische Kritik*, 100:25–50, 1892. Translation in The Frege Reader, ed. M. Beaney, Blackwell Publishers, Oxford, 1997.

[21] Y. Fu. The chi-calculus. In *Proceedings of ICAPDC '97*, pages 74–81. IEEE, Computer Society Press, 1997.

[22] Y. Fu. A proof-theoretical approach to communication. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proceedings of ICALP '97*, volume 1256 of *Lecture Notes in Computer Science*, pages 325–335. Springer-Verlag, 1997.

[23] Y. Fu. Open bisimulations on chi processes. In J. C. M. Baeten and S. Mauw, editors, *Proceedings of CONCUR '99*, volume 1664 of *Lecture Notes in Computer Science*, pages 304–319. Springer-Verlag, 1999.

[24] Y. Fu and Z. Yang. Chi calculus with mismatch. In C. Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 596–610. Springer-Verlag, 2000.

[25] P. Gardner and L. Wischik. Explicit fusions. In M. Nielsen and B. Rovan, editors, *Proceedings of MFCS 2000*, volume 1893 of *Lecture Notes in Computer Science*, pages 373–382. Springer-Verlag, 2000. `http://www.wischik.com/lu/research/explicit-fusions.html`

[26] 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.

[27] O. Group. DCE 1.1: Remote procedure call. Specification C706, Open Group, 1997. `http://www.opengroup.org/onlinepubs/009629399/`

[28] A. Heydon and G. Nelson. The juno-2 constraint-based drawing editor. Research Report 131a, Digital Systems Research Center, 1994. `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-131a.html`

[29] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, New Jersey, 1985.

[30] K. Honda. Elementary structures in process theory (1): Sets with renaming. *Journal of Mathematical Structures in Computer Science*, 10:617–631, 2000. `ftp://ftp.dcs.qmw.ac.uk/lfp/kohei/ps.ps.gz`

[31] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995. `ftp://ftp.dcs.qmw.ac.uk/lfp/kohei/OLD/fst93.ps.gz`

[32] H. Jeffreys. *Scientific Inference*. Cambridge University Press, second edition, 1957.

[33] C. Laneve, J. Parrow, and B. Victor. Solo diagrams. In *Proceedings of TACS 2001*, 2001. To appear. `http://www.docs.uu.se/~victor/tr/solodiagrams.shtml`

[34] C. Laneve and B. Victor. Solos in concert. In J. Wiederman, P. van Emde Boas, and M. Nielsen, editors, *Proceedings of ICALP '99*, volume 1644 of *Lecture Notes in Computer Science*, pages 513–523. Springer-Verlag, 1999. `http://www.docs.uu.se/~victor/tr/solos.shtml`

[35] F. Le Fessant and L. Maranget. Compiling join-patterns. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume16.3.html`

[36] P. J. Leach. *UUIDs and GUIDs*. Microsoft, 1998. `http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt`

[37] J. J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. In C. Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 243–258. Springer-Verlag, 2000. `http://pauillac.inria.fr/~leifer/articles/derbc-concur-final.pdf`

[38] L. Leth and B. Thomsen. Some facile chemistry. *Formal Aspects of Computing*, 7(E):67–110, 1995. `http://www.dcs.gla.ac.uk/~jon/facs/e-papers/`

[39] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[40] M. Merro. On the expressiveness of chi, update, and fusion calculi. In C. Palamidessi and I. Castellani, editors, *Proceedings of EXPRESS '98*, volume 16.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998. `http://www.elsevier.nl/locate/entcs/volume16.2.html`

[41] M. Merro. On equators in asynchronous name-passing calculi without matching. In I. Castellani and B. Victor, editors, *Proceedings of EXPRESS '99*, volume 27 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1999. `http://www.elsevier.nl/locate/entcs/volume27.html`

[42] M. Merro. *Locality in the pi-calculus and applications to distributed objects*. PhD thesis, École des Mines, France, 2000. `http://www.cogs.susx.ac.uk/users/massimo/phdthesis.ps.gz`

[43] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[44] R. Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996. `ftp://ftp.cl.cam.ac.uk/users/rm135/ac9.ps`

[45] R. Milner. *Communicating and mobile systems: the Pi-calculus*. Cambridge University Press, 1999.

[46] R. Milner. Bigraphical reactive systems. In K. G. Larsen and M. Nielsen, editors, *Proceedings of CONCUR 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 16–35. Springer-Verlag, 2001. `http://www.cl.cam.ac.uk/users/rm135/bigraphs.pdf`

[47] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100(1):1–40,41–77, 1992. `http://www.lfcs.informatics.ed.ac.uk/reports/89/ECS-LFCS-89-85/`

[48] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.
`ftp://ftp-sop.inria.fr/mimosa/personnel/davides/bn.ps.gz`

[49] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.
`http://www.acm.org/pubs/toc/Abstracts/0164-0925/69559.html`

[50] G. Nelson and A. Heydon. Juno-2 language definition. Technical Note 1997-009, Digital Systems Research Center, 1997.
`http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-009.html`

[51] J. Parrow and B. Victor. The update calculus. In M. Johnson, editor, *Proceedings of AMAST '97*, volume 1349 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1997.
`http://www.docs.uu.se/~victor/tr/upd.html`

[52] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS '98*, pages 176–185. IEEE, Computer Society Press, 1998.
`http://www.docs.uu.se/~victor/tr/fusion.shtml`

[53] J. Parrow and B. Victor. The tau-laws of fusion. In D. Sangiorgi and R. de Simone, editors, *Proceedings of CONCUR '98*, volume 1466 of *Lecture Notes in Computer Science*, pages 99–114. Springer-Verlag, 1998.
`http://www.docs.uu.se/~victor/tr/taufusion.html`

[54] B. C. Pierce. The pict programming language. homepage.
`http://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html`

[55] B. C. Pierce. Pict: An experiment in concurrent language design. Pict version 3.6 tutorial, University of Edinburgh, 1994.

[56] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing, pages 455–494. MIT Press, 2000.
`http://www.cis.upenn.edu/~bcpierce/papers/pict-design.ps`

[57] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL '98*, pages 378–390. ACM, ACM Press, 1998. `http://portal.acm.org/citation.cfm?id=268978`

[58] D. Sangiorgi. Pi-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(1–2):235–275, 1996.
`http://www.inria.fr/RRRT/RR-2539.html`

[59] D. Sangiorgi. A theory of bisimulation for the pi-calculus. *Acta Informatica*, 33:69–97, 1996.
`ftp://ftp-sop.inria.fr/mimosa/personnel/davides/sub.ps.gz`

[60] D. Sangiorgi and D. Walker. *The Pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[61] J. S. Schwarz. Distributed synchronization of communicating sequential processes. Research Report 56, Department of Artificial Intelligence, University of Edinburgh, 1983.

[62] P. Sewell. On implementations and semantics of a concurrent programming language. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of CONCUR '97*, volume 1243 of *Lecture Notes in Computer Science*, pages 391–405. Springer-Verlag, 1997.
`http://www.cl.cam.ac.uk/~pes20/pict9-crc11pt.ps.gz`

[63] P. Sewell. From rewrite rules to bisimulation congruences. Technical Report 56, Computer Laboratory, University of Cambridge, 1998. To appear in a special issue of TCS for CONCUR '98.
`http://www.cl.cam.ac.uk/~pes20/labels-tcs-final.ps.gz`

[64] P. Sewell, P. Wojciechowski, and B. Pierce. Location-independent communication for mobile agents: a two-level architecture. Technical report, University of Cambridge, 1999.
`http://www.cl.cam.ac.uk/users/pes20/nomadicpict.html`

[65] C. D. Simak. *City*. The Science Fiction Book Club, London, 1952.

[66] W. R. Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall, New Jersey, 1997.

[67] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
`http://portal.acm.org/citation.cfm?id=321884`

[68] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996. `http://www.lfcs.informatics.ed.ac.uk/reports/96/ECS-LFCS-96-345/`

[69] B. Victor. *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes*. PhD thesis, Department of Computer Systems, Uppsala University, Sweden, 1998.
`http://www.docs.uu.se/~victor/thesis.shtml`

[70] B. Victor and J. Parrow. Concurrent constraints in the fusion calculus. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *Lecture Notes in Computer Science*, pages 455–469. Springer-Verlag, 1998.
`http://www.docs.uu.se/~victor/tr/ccfc.shtml`

[71] P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Computer Laboratory, University of Cambridge, 2000.
`http://lsewww.epfl.ch/~pawel/Papers/cl-tr-492.pdf`