The Polyadic π-Calculus: a Tutorial

Robin Milner

Laboratory for Foundations of Computer Science,
Computer Science Department, University of Edinburgh,
The King's Buildings, Edinburgh EH9 3JZ, UK

October 1991

Abstract

The π-calculus is a model of concurrent computation based upon the notion of *naming*. It is first presented in its simplest and original form, with the help of several illustrative applications. Then it is generalized from *monadic* to *polyadic* form. Semantics is done in terms of both a reduction system and a version of labelled transitions called *commitment*; the known algebraic axiomatization of strong bisimilarity is given in the new setting, and so also is a characterization in modal logic. Some theorems about the *replication* operator are proved.

Justification for the polyadic form is provided by the concepts of *sort* and *sorting* which it supports. Several illustrations of different sortings are given. One example is the presentation of data structures as processes which respect a particular sorting; another is the sorting for a known translation of the λ-calculus into π-calculus. For this translation, the equational validity of β-conversion is proved with the help of replication theorems. The paper ends with an extension of the π-calculus to ω-order processes, and a brief account of the demonstration by Davide Sangiorgi that higher-order processes may be faithfully encoded at first-order. This extends and strengthens the original result of this kind given by Bent Thomsen for second-order processes.

This work was done with the support of a Senior Fellowship from the Science and Engineering Research Council, UK.

1 Introduction

The π-calculus is a way of describing and analysing systems consisting of agents which interact among each other, and whose configuration or neighbourhood is continually changing. Since its first presentation [19] it has developed, and continues to do so; but the development has a main stream. In this tutorial paper I give an introduction to the central ideas of the calculus, which can be read by people who have never seen it before; I also show some of the current developments which seem most important – not all of which have been reported elsewhere.

Any model of the world, or of computation (which is part of the world), makes some ontological commitment; I mean this in the loose sense of a commitment as to which phenomena it will try to capture, and which mental constructions are seen to fit these phenomena best. This is obvious for the "denotational" models of computing; for example, the set-theoretic notion of *function* is chosen as the essence or abstract content of the deterministic sequential process by which a result is computed from arguments. But mathematical operations – adding, taking square-roots – existed long before set theory; and it seems that Church in creating the λ-calculus had "algorithm" more in mind than "function" in the abstract sense of the word.

Nevertheless, the λ-calculus makes *some* ontological commitment about computation. It emphasizes the view of computation as taking arguments and yielding results. By contrast, it gives no direct representation of a heterarchical family of agents, each with its changing state and an identity which persists from one computation to another. One may say that the λ-calculus owes its very success to its quite special focus upon argument-result computations.

Concurrent computation, and in particular the power of concurrently active agents to influence each other's activity on the fly, cannot be forced into the "function" mould (set-theoretic or not) without severe distortion. Of course concurrent agents can be assumed (or constrained) to interact in all sorts of different ways. One way would be to treat each other precisely as "function computers"; such an agent's interaction with its environment would consist of receiving arguments and giving results and expecting its sub-agents, computing auxiliary functions, to behave in a similar way. Thus functional computation is a special case of concurrent computation, and we should expect to find the λ-calculus exactly represented within a general enough model of concurrency.

In looking for basic notions for a model of concurrency it is therefore probably wrong to extrapolate from λ-calculus, except to follow its example in seeking something small and powerful. (Here is an analogy: Music is an art form, but it would be wrong to look for an aesthetic theory to cover all art forms by extrapolation from musical theory.) So where else do we look? From one point of view, there is an embarrassingly wide range of idea-sources to choose from; for concurrent computation in the broadest sense is about any co-operative activity among independent agents – even human organizations as well as distributed computing systems. One may even hope that a model of concurrency may attain a breadth of application comparable to physics; Petri expressed such hopes in his seminal work on concurrency [25], and was guided by this analogy.

Because the field is indeed so large, we may doubt whether a single *unified* theory of concurrency is possible; or, even if possible, whether it is good research strategy to seek it so early. Another more modest strategy is to seize upon some single notion which seems to be pervasive, make it the focus of a model, and then submit that model to various tests: Is its intrinsic theory tractable and appealing? Does it apply to enough real situations to be useful in building systems, or in understanding those in existence?

This strategy, at least with a little hindsight, is what led to the π-calculus. The pervasive notion we seize upon is *naming*. One reason for doing so is that naming strongly presupposes *independence*; one naturally assumes that the *namer* and the *named* are co-existing (concurrent) entities. Another reason is that the act of using a name, or address, is inextricably confused with the act of communication. Indeed, thinking about names seems to bring into focus many aspects of computing: problems, if not solutions. If naming is involved in communicating, and is also (as all would agree) involved in locating and modifying data, then we look for a way of treating data-access and communication as the same thing; this leads to viewing data as a special kind of process, and we shall see that this treatment of data arises naturally in the π-calculus.

Another topic which we can hope to understand better through naming is *object-oriented programming*; one of the cornerstones of this topic (which is still treated mostly informally) is the way in which objects provide access to one another by naming. In [17] I used the term *object paradigm* to describe models such as the π-calculus in which agents (objects) are assumed to persist and retain independent identity. David Walker [28] has had initial success in giving formal semantics to simple object-oriented languages in the π-calculus. A challenging problem is to reconcile the assumption, quite common in the world of object-oriented programming, that each object should possess a unique name with the view expressed below (Chapter 1) that naming of *channels*, but not of *agents*, should be primitive in the π-calculus.

By focussing upon naming, we should not give the impression that we expect every aspect of concurrency to be thereby explained. Other focal notions are likely to yield a different and complementary view. Yet naming has a strong attraction (at least for me); it is a notion distilled directly from computing practice. It remains to be seen which intuitions for understanding concurrency will arise from practice in this way, and which will arise directly from logic – which in turn is a distillation of a kind of computational experience, namely inference. Both sources should be heeded. An example of a logical intuition for concurrency is the light cast upon *resource use* by Girard's linear logic [9]. I believe it quite reasonable to view these two sources of intuition as ultimately the same source; then the understanding of computation via naming (say) is just as much a logical activity as is the use of modal logics (say) in computer science.

Background and related work The work on π-calculus really began with a failure, at the time that I wrote about CCS, the Calculus of Communicating Systems [15]. This was the failure, in discussion with Mogens Nielsen at Aarhus in 1979, to see how full mobility among processes could be handled algebraically. The wish to do this was motivated partly by Hewitt's actor systems, which he introduced much earlier [12]. Several years later Engberg and Nielsen [8] succeeded in giving an algebraic formulation. The π-calculus [19] is a simplification and strengthening of their work.

Meanwhile other authors had invented and applied formalisms for processes without the restriction of a finite fixed initial connectivity. Two prominent examples are the DyNe language of Kennaway and Sleep [14], and the work on parametric channels by Astesiano and Zucca [3]. These works are comparable to the π-calculus because they achieve mobility by enriching the handling of *channels*.

By contrast, one can also achieve mobility by the powerful means of transmitting *processes* as messages; this is the *higher-order* approach. It is well exemplified by the work Astesiano and Reggio [2] in the context of general algebraic specification, F. Nielson [22] with emphasis upon type structure, Boudol [6] in the context of λ-calculus, and Thomsen [27]. It has been a deliberate intention in the π-calculus to avoid higher order initially, since the goal was to demonstrate that in some sense it is sufficiently powerful to allow only *names* or *channels* to be the content of communications. Indeed Thomsen's work supports this conjecture, and the present work strengthens his results comparing the approaches. See Milner [17] for a discussion contrasting the approaches.

Outline There are six short chapters following this introduction.

Chapter 2 reviews the formalism of the monadic π-calculus, essentially as it was presented in [19]; it also defines the notion of structural congruence and the reduction relation as first given in [17].

Chapter 3 is entirely devoted to applications; the first defines a simple mobile telephone protocol, the second encodes arithmetic in π-calculus, and the third presents two useful disciplines of name-use (such as may be obeyed in an operating system) in the form of properties invariant under reduction.

Chapter 4 generalizes π-calculus to polyadic communications, introduces the notions of *abstraction* and *concretion* which enhance the power of expression of the calculus (illustrated by a simple treatment of truth values), and affirms that the reduction relation remains essentially unchanged.

Chapter 5 and Chapter 6 provide the technical basis of the work. In Chapter 5, first *reduction congruence* is defined; this is a natural congruence based upon reduction and observability. Next, the standard operational semantics of [19] is reformulated in terms of a new notion, *commitment*; this, together with the flexibility which abstractions and concretions provide, yields a very succinct presentation. Then the (late) bisimilarity of [19] is restated in the polyadic setting, with its axiomatization. Its slightly weaker variant *early* bisimilarity, discussed in

Part II of [19], is shown to induce a congruence identical with reduction congruence. Some theorems about replication are given. Finally, the modal logic of [20], which provides characterizations of both late and early bisimilarity, is formulated in a new way – again taking advantage of the new setting.

Chapter 6 introduces the notions of *sort* and *sorting*, which are somewhat analogous to the simple type hierarchy in λ-calculus, but with significant differences. Data structures are shown to be represented as a particularly well-behaved class of processes, which moreover respect a distinctive *sorting* discipline. Finally, with the help of sorts, new light is cast upon the encoding of λ-calculus into π-calculus first presented in [17]; a simple proof is given of the validity of β-conversion in this interpretation of λ-calculus, using theorems from Chapter 5.

Chapter 7 explores higher-order processes, extending the work of Thomsen [27]. It is shown how sorts and sorting extend naturally not only to second-order (processes-as-data), but even to ω-order; a key rôle is played here by abstractions. A theorem of Sangiorgi [26] is given which asserts that these ω-order processes can be faithfully encoded in the first-order π-calculus (i.e. the calculus of Chapter 4). Some details of this encoding are given.

Acknowledgements I thank Joachim Parrow and David Walker for the insights which came from our original work together on π-calculus, and which have deeply informed the present development. I also thank Davide Sangiorgi and Bent Thomsen for useful discussions, particularly about higher-order processes. I am most grateful to Dorothy McKie for her help and skill in preparing this manuscript.

The work was carried out under a Senior Fellowship funded by the Science and Engineering Research Council, UK.

2 The Monadic π-calculus

2.1 Basic ideas

The most primitive entity in π-calculus is a *name*. Names, infinitely many, are $x, y, \ldots \in \mathcal{X}$; they have no structure. In the basic version of π-calculus which we begin with, there is only one other kind of entity; a *process*. Processes are $P, Q, \ldots \in \mathcal{P}$ and are built from names by this syntax

$$P \quad ::= \quad \Sigma_{i \in I} \pi_i . P_i \quad \Big| \quad P \mid Q \quad \Big| \quad !P \quad \Big| \quad (\nu x)P$$

Here I is a finite indexing set; in the case $I = \emptyset$ we write the sum as $\mathbf{0}$. In a summand $\pi.P$ the prefix π represents an *atomic action*, the first action performed by $\pi.P$. There are two basic forms of prefix:

$x(y)$, which binds y in the prefixed process, means "input some name – call it y – along the link named x",

$\overline{x}y$, which does not bind y, means "output the name y along the link named x".

In each case we call x the *subject* and y the *object* of the action; the subject is *positive* for input, *negative* for output.

A name refers to a link or a channel. It can sometimes be thought of as naming a process at "the other end" of a channel; there is a polarity of names, and $\overline{x}$ – the *co-name* of x – is used for output, while x itself is used for input. But there are two reasons why "naming a process" is not a good elementary notion. The first is that a process may be referred to by many names; it may satisfy different demands, along different channels, for many clients. The second is that a name may access many processes; I may request a resource or a service – e.g. I may cry for help – from any agent able to supply it. In fact, if we had names for processes we would have to have (a different kind of) names for channels too! This would oppose the parsimony which is essential in a basic model.

Of course in human communities it is often convenient, and a convention, that a certain name is borne uniquely by a certain member (as the name "Robin" is borne uniquely by me in my family, but not in a larger community). So, in process communities it will sometimes be a convention that a name x is borne uniquely by a certain process, in the sense that only this member will use the name x as a (positive) subject; then those addressing the process will use the co-name $\overline{x}$ as a (negative) subject. But conventions are not maintained automatically; they require discipline! In fact, that a name is uniquely borne is an invariant which is useful to prove about certain process communities, such as distributed operating systems.

We dwelt at length on this point about naming, because it illustrates so well the point made in the introduction about ontological commitment. We now return to describing the calculus.

The summation form $\Sigma\pi_i.P_i$ represents a process able to take part in one – but only one – of several alternatives for communication. The choice is not made by the process; it can never commit to one alternative until it occurs, and this occurrence precludes the other alternatives. Processes in this form are called *normal processes* (because as we see later, all processes can be converted to this *normal form*). For normal processes $M, N, \ldots \in \mathcal{N}$ we shall use the following syntax:

$$N \ ::= \ \pi.P \ \big| \ \mathbf{0} \ \big| \ M+N$$

In this version of π-calculus we confine summation to normal processes, though previously we have allowed the form $P+Q$ for arbitrary processes. One reason is that the reduction rules in Section 2.4 are simpler with this constraint; another is that forms such as $(P|Q)+R$ have very little significance. However, everything in this paper can be adjusted to allow for the more general use of summation.

What do the last three forms of process mean? $P|Q$ – "P par Q" – simply means that P and Q are concurrently active, so they can act independently – but can also communicate. $!P$ – "bang P" – means $P|P|\ldots$; as many copies as you wish. There is no risk of infinite concurrent activity; our reduction rules will see to that. The operator "!" is called *replication*. A common instance of replication is $!\pi.P$ – a resource which can only be replicated when a requester communicates via π.

Finally, $(\nu x)P$ – "new x in P" – restricts the use of the name x to P. Another way of describing it is that it declares a new unique name x, distinct from all external names, for use in P. The behaviour of (νx) is subtle. In fact, the character of the π-calculus derives from the interplay between its two binding operators: $x(y)$ which binds y somewhat as λy binds y in the λ-calculus, and (νx) which has no exact correlate in other calculi (but is the restriction operator of CCS promoted to a more influential rôle).

Before looking at examples, we introduce a convenient abbreviation. Processes like $x(y).\mathbf{0}$ and $\overline{x}y.\mathbf{0}$ are so common that we prefer to omit the trailing "$.\mathbf{0}$" and write just $x(y)$ and $\overline{x}y$.

2.2 Some simple examples

Consider the process

$$\overline{x}y.\mathbf{0} \mid x(u).\overline{u}v.\mathbf{0} \mid \overline{x}z.\mathbf{0}$$

which we now abbreviate to

$$\overline{x}y \mid x(u).\overline{u}v \mid \overline{x}z$$

Call it $P \mid Q \mid R$. One of two communications (but not both) can occur along the channel x; P can send y to Q, or R can send z to Q. The two alternatives for the result are

$$\mathbf{0} \mid \overline{y}v \mid \overline{x}z \quad \text{or} \quad \overline{x}y \mid \overline{z}v \mid \mathbf{0}$$

Note that R has become $\overline{y}v$ or $\overline{z}v$; thus, the communication has determined which channel R can next use for output, y or z.

Now consider a variant

$$(\nu x)(\overline{x}y \mid x(u).\overline{u}v) \mid \overline{x}z$$

In this case, the (free) x in R is quite different from the (bound) x in P and Q, so only one communication can happen, yielding

$$\mathbf{0} \mid \overline{y}v \mid \overline{x}z$$

(The restriction (νx) has vanished; it has no work left to do, since the x which it restricted has been used up by the communication.)

Third, consider

$$\overline{x}y \mid !x(u).\overline{u}v \mid \overline{x}z$$

This differs from the first case, because Q is now replicated. So $!Q$ can first spin off one copy to communicate with P, and the system becomes

$$\mathbf{0} \mid \overline{y}v \mid !Q \mid \overline{x}z$$

Then $!Q$ can spin off another copy to communicate with R, and the system becomes

$$\mathbf{0} \mid \overline{y}v \mid !Q \mid \overline{z}v \mid \mathbf{0}$$

We have just seen several examples of *reduction*, i.e. the transformation of a process corresponding to a single communication. We now present the π-calculus reduction rules; the analogy with reduction in the λ-calculus is striking but so are the differences.

2.3 Structural Congruence

We have already said that there are two binding operators; the input prefix $x(y)$ (which binds y) and the restriction (νx). So we can define the *free names* $\mathrm{fn}(P)$, and the *bound names* $\mathrm{bn}(P)$ of a process P in the usual way. We extend these to prefixes; note

$$\begin{aligned} \mathrm{bn}(x(y)) = \{y\} \quad &, \quad \mathrm{fn}(x(y)) = \{x\} \\ \mathrm{bn}(\overline{x}y) = \emptyset \quad &, \quad \mathrm{fn}(\overline{x}y) = \{x, y\} \end{aligned}$$

Also, the *names* of a process P are $\mathrm{n}(P) \stackrel{\mathrm{def}}{=} \mathrm{bn}(P) \cup \mathrm{fn}(P)$.

Now, to make our reduction system simple, we wish to identify several expressions. A typical case is that we want $+$ and $\mid$ to be commutative and associative. We therefore define *structural congruence* $\equiv$ to be the smallest congruence relation over $\mathcal{P}$ such that the following laws hold:

1. Agents (processes) are identified if they only differ by a change of bound names

2. $(\mathcal{N}/\equiv, +, \mathbf{0})$ is a symmetric monoid

3. $(\mathcal{P}/\equiv, |, \mathbf{0})$ is a symmetric monoid

4. $!P \equiv P \mid !P$

5. $(\nu x)\mathbf{0} \equiv \mathbf{0}, (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$

6. If $x \notin \mathrm{fn}(P)$ then $(\nu x)(P|Q) \equiv P \mid (\nu x)Q$

Exercise Use 3, 5 and 6 to show that $(\nu x)P \equiv P$ when $x \notin \mathrm{fn}(P)$. ∎

Note that laws 1, 4 and 6 allow any restriction not inside a normal process to be pulled into outermost position; for example, if $P \equiv (\nu y)\overline{x}y$ then

$$\begin{aligned} x(z).\overline{y}z \mid !P &\equiv x(z).\overline{y}z \mid (\nu y)\overline{x}y \mid !P \\ &\equiv x(z).\overline{y}z \mid (\nu y')\overline{x}y' \mid !P \\ &\equiv (\nu y')(x(z).\overline{y}z \mid \overline{x}y') \mid !P \end{aligned}$$

This transformation has brought about the juxtaposition $x(z).\cdots \mid \overline{x}y'.\cdots$, which is reducible by the rules which follow below. The use of structural laws such as the above, to bring communicands into juxtaposition, was suggested by the Chemical Abstract Machine of Berry and Boudol [5].

2.4 Reduction rules

This section is devoted to defining the *reduction relation* $\rightarrow$ over processes; $P \rightarrow P'$ means that P can be transformed into P' by a single computational step. Now every computation step consists of the interaction between two normal terms. So our first reduction rule is *communication*:

$$\textsc{comm}: \ (\cdots + x(y).P) \mid (\cdots + \overline{x}z.Q) \rightarrow P\{z/y\} \mid Q$$

There are two ingredients here. The first is how communication occurs between two atomic normal processes $\pi.P$ which are complementary (i.e. whose subjects are complementary). The second is the discard of alternatives; either instance of "$\cdots$" can be $\mathbf{0}$ of course, but if not then the communication pre-empts other possible communications.

COMM is the only axiom for $\rightarrow$; otherwise we only have inference rules, and they are three in number. The first two say that reduction can occur underneath composition and restriction, while the third simply says that structurally congruent terms have the same reductions.

$$\textsc{par}: \ \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad\qquad \textsc{res}: \ \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

$$\textsc{struct}: \ \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

Exercise In Section 2.2 and the previous exercise several reductions were given informally. Check that they have all been inferred from the four rules for $\rightarrow$. ∎

It is important to see what the rules do *not* allow. First, they do not allow reductions underneath prefix, or sum; for example we have

$$u(v).(x(y) \mid \overline{x}z) \not\rightarrow$$

Thus prefixing imposes an order upon reduction. This constraint is not necessary. However, the calculus changes non-trivially if we relax it, and we shall not consider the possibility further in this paper.

Second, the rules do not allow reduction beneath replication. In some sense, this does not reduce the computational power; for if we have $P \rightarrow P'$ then, instead of inferring $!P \rightarrow !P'$, which is equivalent to allowing unboundedly many coexisting copies of P to reduce, we can always infer

$$!P \equiv \underbrace{P \mid P \mid \cdots \mid P}_{n \text{ times}} \mid !P \rightarrow^n P' \mid P' \mid \cdots \mid P' \mid !P$$

thus (in n reductions) reducing as many copies of P as we require – and for finite work we can only require finitely many !

Third, the rules tell us nothing about *potential* communication of a process P with other processes. From the reduction behaviour alone of P and Q separately, we cannot infer the whole reduction behaviour of, say, $P|Q$. (This is just as in the λ-calculus, where λxx and λxxx have the *same* reduction behaviour – they have no reductions – but applying them to the same term λyy gives us two terms $(\lambda xx)(\lambda yy)$ and $(\lambda xxx)(\lambda yy)$ with *different* reduction behaviour.)

If we wish to identify *every* potential communication of a process, so as to distinguish say $\overline{x}y$ from $\overline{x}z$, then we would indeed become involved with the familiar labelled transition systems used in process algebra (and introduced later in this paper). We do not want to do this yet. But for technical reasons we want to do a little of it. To be precise, we only want to distinguish processes which can perform an external communication at some location α – a name or co-name – from those which cannot. So we give a few simple definitions.

First, we say that Q occurs *unguarded* in P if it occurs in P but not under a prefix. Thus, for example, Q is unguarded in $Q|R$ and in $(\nu x)Q$ but not in $x(y).Q$. Then we say P is *observable at* α – and write $P \downarrow_\alpha$ – if some $\pi.Q$ occurs unguarded in P, where α is the subject of π and is unrestricted. Thus $x(y) \downarrow_x$ and $(\nu z)\overline{x}z \downarrow_{\overline{x}}$, but $(\nu x)\overline{x}z \not\downarrow_{\overline{x}}$; also $(\nu x)(x(y) \mid \overline{x}z) \not\downarrow_x$ even though it has a reduction.

It turns out that we get an interesting congruence over $\mathcal{P}$ in terms of $\rightarrow$ and $\downarrow_\alpha$. This will be set out in Chapter 4; first we digress in Chapter 3 to look at several applications.

3 Applications

In this section, we give some simple illustrations of the π-calculus. We begin by introducing a few convenient derived forms and abbreviations.

3.1 Some derived forms

In applications, we often want forms which are less primitive than the basic constructions of monadic π-calculus. One of the first things we find useful is multiple inputs and outputs along the same channel. A natural abbreviation could be to write e.g. $x(yz)$ for $x(y).x(z)$ and $\overline{x}yz$ for $\overline{x}y.\overline{x}z$. But this would give a misleading impression about the indivisibility of the pair of actions in each case. Consider

$$x(yz) \mid \overline{x}y_1z_1 \mid \overline{x}y_2z_2$$

for example; the intention is that y, z should get bound to either y_1, z_1 or y_2, z_2. But if we adopt the above abbreviations there is a third possibility, which is a mix-up; y, z can get bound to y_1, y_2. To avoid this mix-up, a way is needed of making a *single* commitment to any multiple communication, and this can be done using private (i.e. restricted) names. So we introduce abbreviations

$$\begin{array}{rl} x(y_1 \cdots y_n) & \text{for} \quad x(w).w(y_1).\cdots.w(y_n) \\ \overline{x}y_1 \cdots y_n & \text{for} \quad (\nu w)\overline{x}w.\overline{w}y_1.\cdots.\overline{w}y_n \end{array}$$

– writing just x for $x()$ when $n = 0$. You can check that the mix-up in the example is no longer possible. The abbreviation has introduced an extra communication, even in the case $n = 1$, but this will cause no problem.

Next, we often wish to define parametric processes recursively. For example, we may like to define A and B, of arity 1 and 2 respectively, by

$$A(x) \stackrel{\text{def}}{=} x(yz).B(y,z) \quad , \quad B(y,z) \stackrel{\text{def}}{=} \overline{y}z.A(z)$$

If we wish to allow such parametric process definitions of the general form $K(\vec{x}) \stackrel{\text{def}}{=} P_K$, we add

$$P \quad ::= \quad \cdots \quad \Big| \quad K(\vec{y})$$

to the syntax of processes, where K ranges over process identifiers; for each definition we also add a new structural congruence law $K(\vec{y}) \equiv P_K\{\vec{y}/\vec{x}\}$ to those given in Section 2.3.

However, it is easier to develop a theory if "definition-making" does not have to be taken as primitive. In fact, provided the number of such recursive definitions is finite, we can encode them by replication; then the introduction of new constants, with definitions, is just a matter of convenience. We shall content ourselves with showing how to encode a single recursive definition with a single parameter. Thus, suppose we have

$$A(x) \stackrel{\text{def}}{=} P$$

where we assume that $\mathrm{fn}(P) \subseteq \{x\}$, and that P may contain occurrences of A (perhaps with different parameters). The idea is, first, to replace every recursive call $A(y)$ within P by a little process $\overline{a}y$ which excites a new copy of P. (Here a is a new name.) Let us denote by $\widehat{P}$ the result of doing these replacements in P. Then the replication

$$!a(x).\widehat{P}$$

corresponds to the parametric process $A(x)$. We now have to take care of the outermost calls of A. So let $A(z)$ occur in some system S; then we replace it by

$$(\nu a)(\overline{a}z \mid !a(x).\widehat{P})$$

Note that this places a separate copy of the replication at each call $A(z)$ in S. Alternatively one can make do with a single copy; transform S to $\widehat{S}$ by replacing each call $A(z)$ just by $\overline{a}z$, and then replace S by

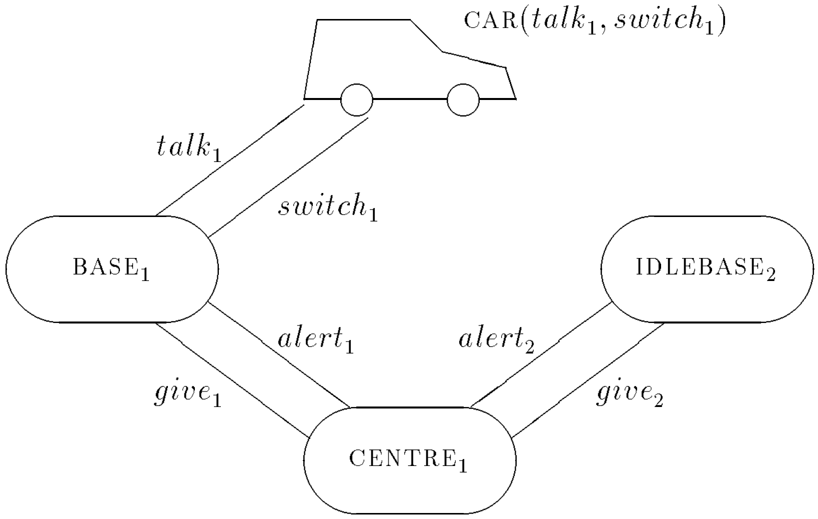
$$(\nu a)(\widehat{S} \mid !a(x).\widehat{P})$$

Of course, these translations do not behave identically with the original, because they do one more reduction for each call of A; but they are *weakly congruent* to the original (in the sense of [19]), which is all we would require in applications.

From now on, in applications we shall freely use parametric recursive definitions; but, knowing that translation is possible, in our theoretical development we shall ignore them and stick to replication.

3.2 Mobile telephones

Here is a "flowgraph" of our first application:



This is a simplified version of a system used by Orava and Parrow [23] to illustrate π-calculus. A CENTRE is in permanent contact with two BASE stations, each in a different part of the country. A CAR with a mobile telephone moves about the country; it should always be in contact with a BASE. If it gets rather far from its current BASE contact, then (in a way which we do not model) a hand-over procedure is initiated, and as a result the CAR relinquishes contact with one BASE and assumes contact with another.

The flowgraph shows the system in the state where the CAR is in contact with BASE_1; it may be written

$$\begin{aligned}\mathrm{SYSTEM}_1 \stackrel{\mathrm{def}}{=} & (\nu\, talk_i, switch_i, give_i, alert_i \ : \ i=1,2)\\ & \big(\mathrm{CAR}(talk_1, switch_1) \mid \mathrm{BASE}_1 \mid \mathrm{IDLEBASE}_2 \mid \mathrm{CENTRE}_1\big)\end{aligned}$$

What about the components?

A CAR is parametric upon a *talk* channel and a *switch* channel. On *talk* it can talk repeatedly; but at any time along *switch* it may receive two new channels which it must then start to use:

$$\begin{aligned}\mathrm{CAR}(talk, switch) \stackrel{\mathrm{def}}{=} & \ talk \,.\, \mathrm{CAR}(talk, switch)\\ & + \ switch(talk' switch') \,.\, \mathrm{CAR}(talk', switch')\end{aligned}$$

A BASE can talk repeatedly with the CAR; but at any time it can receive along its *give* channel two new channels which it should communicate to the CAR, and then become idle itself; we define

$$\begin{aligned}\mathrm{BASE}(t,s,g,a) \stackrel{\mathrm{def}}{=} & \ t \,.\, \mathrm{BASE}(t,s,g,a)\\ & + g(t's') \,.\, \overline{s}t's' \,.\, \mathrm{IDLEBASE}(t,s,g,a)\end{aligned}$$

An IDLEBASE, on the other hand, may be told on its *alert* channel to become active:

$$\mathrm{IDLEBASE}(t,s,g,a) \stackrel{\mathrm{def}}{=} a \,.\, \mathrm{BASE}(t,s,g,a)$$

We define the abbreviation

$$\mathrm{BASE}_i \stackrel{\mathrm{def}}{=} \mathrm{BASE}(talk_i, switch_i, give_i, alert_i) \qquad (i=1,2)$$

and a similar abbreviation $\mathrm{IDLEBASE}_i$. Thus, for example,

$$\begin{aligned}\mathrm{BASE}_i &\equiv talk_i \,.\, \mathrm{BASE}_i + give_i(t's') \,.\, \overline{switch_i}t's' \,.\, \mathrm{IDLEBASE}_i\\ \mathrm{IDLEBASE}_i &\equiv alert_i \,.\, \mathrm{BASE}_i\end{aligned}$$

Finally the CENTRE, which initially knows that the CAR is in contact with BASE_1, can decide (according to information which we do not model) to transmit the channels $talk_2$, $switch_2$ to the CAR via BASE_1, and alert BASE_2 of this fact. So we define

$$\begin{aligned}\mathrm{CENTRE}_1 &\stackrel{\mathrm{def}}{=} \overline{give_1} talk_2 switch_2 \,.\, alert_2 \,.\, \mathrm{CENTRE}_2\\ \mathrm{CENTRE}_2 &\stackrel{\mathrm{def}}{=} \overline{give_2} talk_1 switch_1 \,.\, alert_1 \,.\, \mathrm{CENTRE}_1\end{aligned}$$

Exercise Check carefully that indeed SYSTEM_1 reduces in three steps to SYSTEM_2, which is precisely SYSTEM_1 with the subscripts 1 and 2 interchanged. The reduction is (using $\vec{c}$ for the set of eight restricted channels):

$$\begin{array}{rcl}
\text{SYSTEM}_1 & \equiv & (\nu\vec{c})\Big(\text{CAR}(talk_1, switch_1) \mid \text{BASE}_1 \mid \text{IDLEBASE}_2 \mid \text{CENTRE}_1\Big) \\
& \rightarrow & (\nu\vec{c})\Big(\text{CAR}(talk_1, switch_1) \mid \overline{switch}_1 talk_2 switch_2 \,.\, \text{IDLEBASE}_1 \\
& & \qquad \mid \text{IDLEBASE}_2 \mid alert_2 \,.\, \text{CENTRE}_2\Big) \\
& \rightarrow & (\nu\vec{c})\Big(\text{CAR}(talk_2, switch_2) \mid \text{IDLEBASE}_1 \\
& & \qquad \mid \text{IDLEBASE}_2 \mid alert_2 \,.\, \text{CENTRE}_2\Big) \\
& \rightarrow & (\nu\vec{c})\Big(\text{CAR}(talk_2, switch_2) \mid \text{IDLEBASE}_1 \mid \text{BASE}_2 \mid \text{CENTRE}_2\Big) \\
& \equiv & \text{SYSTEM}_2
\end{array}$$

■

Of course this example is highly simplified. Consider one possible refinement. There is no reason why the number of available (*talk, switch*) channel-pairs is equal to the number of BASES; nor that each base always uses the same channel-pair. The reader may like to experiment with having an arbitrary (fixed) number of BASES; at each handover the new BASE could be chosen at random, and a channel-pair picked from a store of available channel pairs maintained (say) in a queue.

3.3 Numerals and arithmetic

For our second application we show that arithmetic can be done in π-calculus in much the same way as it can in λ-calculus. Church represented the natural number n in λ-calculus by

$$\lambda f \lambda x f^n(x)$$

– i.e. the function which iterates its function argument n times.

As a first attempt in π-calculus, we may choose to represent n by the parametric process

$$\underline{n}(x) \stackrel{\text{def}}{=} \underbrace{\overline{x}.\cdots.\overline{x}}_{n \text{ times}}$$

which we abbreviate to $(\overline{x}.)^n$. But this process cannot be tested for zero, and the arithmetic operators (coded also as processes) will need a test for zero.

So we give $\underline{n}$ two parameters, one representing successor, and the other representing zero:

$$\underline{n}(xz) \stackrel{\text{def}}{=} (\overline{x}.)^n\, \overline{z}$$

Now, how do we do arithmetic? We shall represent binary summation, for example, by a parametric process $Add(x_1z_1, x_2z_2, yw)$; the channels x_i, z_i represent the arguments and y, w represent the result. (The commas separating the six parameters of Add are just for clarity.) The correctness of this representation is expressed by the equation

$$(\nu x_1z_1x_2z_2)\Big(\underline{n_1}(x_1z_1) \mid \underline{n_2}(x_2z_2) \mid Add(x_1z_1, x_2z_2, yw)\Big) \;\approx\; \underline{n_1{+}n_2}(yw)$$

where $\approx$ means weak congruence. To achieve this, we first define "copy" and "successor" by mutual recursion:

$$\begin{aligned} Copy(xz, yw) &\stackrel{\text{def}}{=} x.Succ(xz, yw) + z.\overline{w} \\ Succ(xz, yw) &\stackrel{\text{def}}{=} \overline{y}.Copy(xz, yw) \end{aligned}$$

Now, though we have not developed the machinery here, one can easily prove by induction on n that

$$\begin{aligned} (\nu xz)\Big(\underline{n}(xz) \mid Copy(xz, yw)\Big) &\approx \underline{n}(yw) \\ (\nu xz)\Big(\underline{n}(xz) \mid Succ(xz, yw)\Big) &\approx \underline{n{+}1}(yw) \end{aligned}$$

Consider the induction step for example; we have

$$\begin{aligned} (\nu xz)\Big(\underline{n{+}1}(xz) \mid Copy(xz, yw)\Big) &\rightarrow (\nu xz)\Big(\underline{n}(xz) \mid \overline{y}.Copy(xz, yw)\Big) \\ &\approx \overline{y}.(\nu xz)\Big(\underline{n}(xz) \mid Copy(xz, yw)\Big) \\ &\approx \overline{y}.\underline{n}(yw) \quad \text{(by induction)} \\ &\equiv \underline{n{+}1}(yw) \end{aligned}$$

(One step in this argument needs justification: the extraction of $\overline{y}$ into leading position. Also, to complete the argument, one has to show why the reduction $\rightarrow$ in the first line can be replaced by $\approx$. These are routine consequences of the theory of weak congruence.)

Finally, having $Copy$ available, we can define addition by

$$Add(x_1z_1, x_2z_2, yw) \;\stackrel{\text{def}}{=}\; x_1.\overline{y}.Add(x_1z_1, x_2z_2, yw) + z_1.Copy(x_2z_2, yw)$$

Exercise Show that

$$(\nu x_1z_1x_2z_2)\Big(\underline{n}(x_1z_1) \mid \underline{0}(x_1z_2) \mid Add(x_1z_1x_2z_2, yw)\Big) \;\approx\; \underline{n}(yw)$$

by induction, using a similar argument to the inductive proof given earlier. Then prove the general correctness property for Add. ■

The reader will have noticed that numerals are ephemeral; $\underline{n}(xz)$ can only be accessed once. This is why copying is needed. In fact, if you try to define multiplication you will find that you first need something like $Double(xz, y_1w_1, y_2w_2)$ which produces two copies of $\underline{n}(xz)$ from one. It is natural to expect replication to come to our aid. In fact, replication cannot be used directly with the numerals as defined above, but it can be used with the more general representation of data structures which we treat later.

3.4 Invariants in process communities

We now look at two desirable behavioural properties of mobile systems, and we are able to find syntactic conditions which ensure that these properties are enjoyed.

Unique names In Section 2.1 we mentioned the unique naming discipline. Now we formalize it in a simple way, and give conditions under which a process community obeys the discipline.

Consider a system

$$S \quad \equiv \quad (\nu \vec{v})(P_1 \mid \cdots \mid P_m \mid !N_1 \mid \cdots \mid !N_n)$$

We call each P_i and $!N_j$ a *component* of S. We say that S is *friendly* (there is no obvious word to use!) if it satisfies the following conditions:

1. $\mathrm{fn}(S) = \emptyset$, i.e. S is a closed system.

2. No P_i or N_j contains a composition or a replication.

Many systems are friendly; for example, the mobile telephone system in Section 3.2 is friendly when its recursion is coded into replication. The first thing to notice about them is that they *stay* friendly, that is, friendliness is preserved by reduction. This is easy to prove. Note, however, reduction can change m; a friendly system works by spinning off copies of its replications as often as required – and each replica becomes a P_i.

Now, let us say that any process P *bears the name* x if x occurs free in P as a positive subject. This is clearly a necessary condition for P to receive input immediately at x. (It is also sufficient, if the occurrence is unguarded.) For example, $!N_j$ bears x if $N_j \equiv x(y).P$; then sending a message along x, i.e. "addressing the component by name", will spin off a copy of the resource $!N_j$. One thing we would ensure for a friendly system is that, at any one time, at most one component – whether P_i or $!N_j$ – can receive a message along x. Now consider the condition on S that

3. At most one component bears x.

This clearly means that any output along x (from another component) has a determined destination. But condition (3) is not preserved by reduction! For one thing, a replication may well produce two components bearing x. Also, a component P_i may *acquire* the bearing of x; for example if

$$\begin{array}{rcl} P_1 \mid P_2 & \equiv & v(z).z(w) \mid \overline{v}x.x(w) \\ & \rightarrow & x(w) \mid x(w) \end{array}$$

then clearly condition (3) is destroyed by the reduction. This gives us a hint about what extra conditions will ensure preservation of (3):

4. If $N_j \equiv \Sigma\pi_{jk}.Q_{jk}$, then no Q_{jk} bears x.

5. For any v and z, if the expression $v(z).P$ occurs anywhere in S then P does not bear z.

Note that (4) still allows N_j itself to bear x. Condition (5) says, in effect, that the bearing of a name can never be acquired.

It is not quite obvious, but can be proved, that conditions (1–5) together are invariant under reduction.

Now let us say that a name x is *uniquely borne* in S if, whenever $S \rightarrow^* S'$, then S' satisfies condition (3). Intuitively, this means that output along x will *always* have a determined destination. We have been able to give syntactic conditions, (1–5), which ensure that a given x is uniquely borne in a given system S.

Unique handles There is a dual property to unique naming, which it is useful to ensure in an operating system – even in a sequential one, and *a fortiori* in a concurrent one! This is the property that, at any one time, only one component can handle a given resource. To make things precise, let us say that P *can handle* the name x if x occurs free in P as a negative subject. This is clearly a necessary condition for P to be able to send output immediately along x – and also sufficient, if the occurrence is unguarded.

Now we look at friendly systems S, as before, and we consider the condition on S that

6. At most one component can handle x.

This means, for example, that a sequence of messages along x from two different components will not be accidentally interleaved; this is clearly desirable if x gives access to a printing device. Again, condition (6) is not preserved by reduction; this can be argued just as for condition (3) above.

But now we wish to proceed differently from unique naming. In that case, our conditions ensured that the *bearing* of a name could never be acquired. Here, by contrast, we want to allow the *handling* capability to be transmitted freely among components, subject to (6). Therefore we do not wish to impose a condition on inputs, like (5). Instead, we want to impose a condition on *outputs*; when a component transmits a handle, it should neither use it again nor transmit it again.

We naturally arrive at the following two conditions:

7. No N_j can handle x.

8. For any v and z, if $\overline{v}z.P$ occurs anywhere in S then P does not contain z free.

Condition (8) says, in effect, that when an agent transmits a name then it must "forget" it. Note that z may be bound in S; that is, it may stand for a name – perhaps x itself – to be received at some time from another component.

Now, let us say that a name x is *uniquely handled* in S if, whenever $S \rightarrow^* S'$, then S' satisfies condition (6). Then the syntactic conditions (1, 2, 6–8) ensure that a given x is uniquely handled in a given system S.

Exercise We have concentrated on simplicity in giving our conditions. Try to find slightly weaker conditions which ensure that a name is uniquely borne, or uniquely handled, in a friendly system. ∎

The two examples we have just looked at suggest that the syntax of π-calculus is rich enough to allow many interesting structural invariants to be defined, which in turn ensure useful behavioural properties. In both cases, the reader may have felt that the blanket condition on *all* inputs in (5), or *all* outputs in (8), was unnecessarily strong. But it can be weakened very satisfactorily to apply only to names of a particular *sort*, when we have introduced our sorting discipline in Chapter 6.

4 The Polyadic π-calculus

4.1 Abstractions

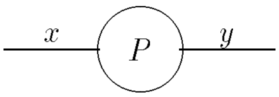
In Section 3.1 we saw that a polyadic input $x(y_1 \cdots y_n)$, or polyadic output $\overline{x}y_1 \cdots y_n$, can be encoded quite straightforwardly in the monadic π-calculus. So at first sight we may regard polyadicity as a mere abbreviational device, with no theoretical interest. But for more than one reason, we shall gain by taking polyadic communication as primitive.

The first reason is to do with abstractions. (The second reason is to do with sorts, and will be deferred to Chapter 6.) An *abstraction* takes the form $(\lambda x_1 \cdots x_n)P$ or equivalently $(\lambda x_1) \cdots (\lambda x_n)P$; it is just an abstraction of names from a process. It is quite different from abstraction in λ-calculus, because a bound name will only ever be instantiated to a name – never to a compound term.

Abstractions are useful in various ways. First, they are the essence of parametric definition; instead of writing $K(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$, we naturally write

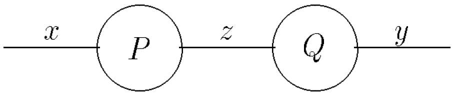
$$K \stackrel{\text{def}}{=} (\lambda x_1 \cdots x_n)P$$

A second use for abstraction is in defining combinators. Consider for example a process with two free names x and y, representing links:



Now if we want to chain together several such processes, with the y link of one joining the x link of its right neighbour, in CCS we would use the renaming operator $[z/x]$ to define the chaining combinator, $\frown$, thus:

$$P \frown Q \stackrel{\text{def}}{=} (\nu z)(P[z/y] \mid Q[z/x])$$



Now abstraction in the π-calculus renders the renaming operator superfluous. For in fact we define the chaining combinator $\frown$ not over processes but over (binary) abstractions:

$$F \frown G \stackrel{\text{def}}{=} (\lambda xy)(\nu z)(Fxz \mid Gzy)$$

Indeed, suppose $F \equiv (\lambda xy)P$, and $G \equiv (\lambda xy)Q$; then we obtain easily

$$(F \frown G)xy \equiv (\nu z)(P\{z/y\} \mid Q\{z/x\})$$

where of course substitution $\{z/x\}$ is a *meta*-syntactic operator.

Thus we see that abstraction is a handy *definitional* device in π-calculus ; but we should remain clear that it has none of the *computational* significance which it possesses in the λ-calculus because this significance depends upon instantiating a bound variable to an arbitrary term – and this is grammatically incorrect in the π-calculus! [1]

Now since abstraction earns a place on its own merit, it would be ridiculous to preserve its distinction from the other binding operator for which instantiation is allowed, namely input prefix. For we can simply declare the abbreviation

$$x(y).P \stackrel{\text{def}}{=} x.(\lambda y)P$$

thus factoring input prefix into two parts; one part is abstraction, and the other we shall call *location*. In the above, x is the location of $(\lambda y)P$; it indicates *where* the name (for which y stands) should be received. But then, of course, it is quite unreasonable not to allow polyadic input:

$$x(y_1 \cdots y_n).P \stackrel{\text{def}}{=} x.(\lambda y_1 \cdots y_n)P$$

[1] A notational question arises. Adopting the form (λx) for abstraction, we risk the misconception that the π-calculus is an extension of the λ-calculus. This is a serious misconception, because one of the main motivations for the π-calculus has been the belief that, in order to unify functional and concurrent computation, we needed basic constructions *more primitive* than those in λ-calculus! In some ways the lightweight form (x), used by Martin-Löf and others, would have been more attractive. But with two forms of binding, abstraction and restriction, it seems clearer to mark each with a symbol. Also, in the higher-order processes in Chapter 7 abstraction of process-variables is used; this is closer to λ-calculus, so the symbol λ is more appropriate, and it is smoother to have one notation for abstraction in all versions of π-calculus.

More than this; when we study sorts, we shall find that it is not only *reasonable*, but actually *necessary*, to allow polyadic input if we wish to respect a very natural sort discipline.

We shall say that the abstraction $(\lambda y_1 \cdots y_n)P$ has *arity* n. In particular, a process P is an abstraction with zero arity.

4.2 Concretions

We would like to treat output dually to input. This immediately suggests that we consider the output prefix form as a derived form:

$$\overline{x}y_1 \cdots y_n.P \quad \stackrel{\mathrm{def}}{=} \quad \overline{x}.[y_1 \cdots y_n]P$$

thus factoring it into two parts: the *co-location* $\overline{x}$, and the *concretion* $[y_1 \cdots y_n]P$. We consider this equivalent to $[y_1] \cdots [y_n]P$, and its arity is n; we call each y_i a *datum-name* of the concretion. Any process P is a concretion with zero arity. Some abbreviations are convenient; we write $\overline{x}.[\vec{y}]$ for $\overline{x}.[\vec{y}]\mathbf{0}$, $\overline{x}.P$ for $\overline{x}.[]P$ and $\overline{x}$ for $\overline{x}.[]$.

(It may be useful to consider concretions $C, D, \ldots$ as having arity ≤ 0; then an *agent* – either an abstraction or a concretion – may have any integer as arity.)

Now consider the simple form of reduction

$$x(\vec{y}).P \mid \overline{x}\vec{z}.Q \quad \rightarrow \quad P\{\vec{z}/\vec{y}\} \mid Q$$

where $\vec{y}, \vec{z}$ are name vectors of equal length (the components of $\vec{y}$ being distinct). This is the natural generalization of COMM to the polyadic case (ignoring + for the moment). It now takes the form

$$x.F \mid \overline{x}.C \rightarrow F \bullet C$$

where F and C are an abstraction and concretion of equal arity, and the pseudo-application $-\bullet-$ is defined in an obvious way (see Section 4.3 below).

So far, concretions may appear to be no more than notational elegance. In fact, they have some conceptual significance. We can illustrate this by considering truth-values and case-analysis: this will also illustrate the importance of admitting the use of restriction (νx) upon concretions.

In λ-calculus, the terms $\lambda x \lambda y x$ and $\lambda x \lambda y y$ are often taken to represent the two truth-values. Now that we have abstraction, we can analogously define

$$\begin{aligned} \mathit{True} &\stackrel{\mathrm{def}}{=} (\lambda t)(\lambda f)\overline{t} \\ \mathit{False} &\stackrel{\mathrm{def}}{=} (\lambda t)(\lambda f)\overline{f} \end{aligned}$$

in π-calculus. We would often have a truth-value *located* at a boolean location b, e.g. the process $b.\mathit{True}$. This located truth-value may be compared with the natural number $\underline{n}(xz)$ in Section 3.3, which was located by a pair of names; but with the introduction of abstractions we now have a purer representation, for *unlocated* data, and indeed we shall extend this to general data structures later.

Now consider the following concretion, representing *case-analysis*:

$$Cases(P,Q) \stackrel{\text{def}}{=} [tf](t.P+f.Q)$$

where t and f do not occur in P and Q. It represents the offer to select P or Q, by using the name t or f. In fact, using pseudo-application, we find

$$\begin{aligned} True \bullet Cases(P,Q) &\equiv \bar{t} \mid (t.P+f.Q) \\ &\rightarrow P \end{aligned}$$

and similarly $False \bullet Cases(P,Q) \rightarrow Q$. This is still slightly imperfect, for we should prefer $Cases(P,Q)$ to have no more free names than P and Q. We therefore prefer to represent case-analysis by a *restricted* concretion:

$$Cases(P,Q) \stackrel{\text{def}}{=} (\nu tf)[tf](t.P+f.Q)$$

Moreover this ensures that t and f are distinct names. Now, we have instead

$$\begin{aligned} True \bullet Cases(P,Q) &\equiv (\nu tf)(\bar{t} \mid (t.P+f.Q)) \\ &\approx P \end{aligned}$$

Note that the right-hand side not only reduces to P, but is weakly congruent ($\approx$) to P, because of the restriction.

To emphasize the role played by concretions, consider the familiar conditional form

$$\texttt{if } b \texttt{ then } P \texttt{ else } Q$$

In π-calculus we take it to mean "inspect the truth-value located at b and perform P or Q according to the value". Now, we can see the conditional form as just the *co-located* case-analysis

$$\bar{b}.Cases(P,Q)$$

and indeed we have

$$b.True \mid \bar{b}.Cases(P,Q) \;\rightarrow\; True \bullet Cases(P,Q) \;\approx\; P$$

An intriguing point is that abstraction and (restricted) concretion offer two subtly different forms of closure for an arbitrary process P. For if the free names of P are $\vec{x}$, then we may call $(\lambda\vec{x})P$ the *abstract* closure, and $(\nu\vec{x})[\vec{x}]P$ the *concrete* closure. They differ in this sense: the concrete closure ensures that P's free names are all distinct from each other, and distinct from all names in the environment; on the other hand, the abstract closure offers arbitrary instantiation of these names.

Finally, why should we not allow mixed abstraction and concretion, such as $(\lambda x)[y]P$ or $[y](\lambda x)P$? To allow this does indeed enrich the calculus in a valuable way. But we leave it to a future paper; the present version of the calculus is a natural whole, and rich enough for our present purposes.

4.3 Syntax, structural congruence and reduction

In moving from the monadic to the polyadic calculus, the main differences are that prefixes $x(\vec{y})$ and $\overline{x}\vec{y}$ are no longer primitive, but become abbreviations, and that we add the forms for abstractions $F, G, \ldots$ and concretions $C, D, \ldots$, calling them collectively *agents* $A, B, \ldots$. We use $\alpha, \beta, \ldots$ to range over names and co-names, and $\vec{x}, \vec{y}, \ldots$ to stand for vectors of names, with length $|\vec{x}|, |\vec{y}|, \ldots$.

$$\begin{array}{rrcl}
\text{Normal processes :} & N & ::= & \alpha.A \;\big|\; \mathbf{0} \;\big|\; M+N \\
\text{Processes :} & P & ::= & N \;\big|\; P \mid Q \;\big|\; !P \;\big|\; (\nu x)P \\
\text{Abstractions :} & F & ::= & P \;\big|\; (\lambda x)F \;\big|\; (\nu x)F \\
\text{Concretions :} & C & ::= & P \;\big|\; [x]C \;\big|\; (\nu x)C \\
\text{Agents :} & A & ::= & F \;\big|\; C
\end{array}$$

Over this syntax, we again wish to represent those identifications which have no computational significance by structural congruence, $\equiv$. For processes, the laws (1–6) for structural congruence are just those given in Section 2.3, understanding that change of bound names is allowed in any agent whatever. Then we add the following rules:

7. $(\nu y)(\lambda x)F \equiv (\lambda x)(\nu y)F \quad (x \neq y)$
8. $(\nu y)[x]C \equiv [x](\nu y)C \quad (x \neq y)$
9. $(\nu x)(\nu y)A \equiv (\nu y)(\nu x)A \quad , \quad (\nu x)(\nu x)A \equiv (\nu x)A$

Some interesting consequences of these laws are not immediately obvious. We can, in fact, use laws 1 and 7 to convert every abstraction F to a *standard form*

$$F \;\equiv\; (\lambda\vec{x})P$$

by pushing restrictions inwards. We cannot do the same for concretions, because the datum-name y is *free* in $[y]C$. But we can use laws 1 and 8 to pull all restrictions of data-names outwards, and push all other restrictions inwards, yielding a *standard form*

$$C \;\equiv\; (\nu\vec{y})[\vec{x}]P \qquad (\vec{y} \subseteq \vec{x})$$

The arity of the abstraction F or concretion C is just the length of the vector $\vec{x}$ in its standard form.

We now define *pseudo-application* $F{\bullet}C$ of an abstraction to a concretion, confining ourselves to the case in which F and C have equal arity. Let $F \equiv (\lambda\vec{x})P$ and $C \equiv (\nu\vec{z})[\vec{y}]Q$ where $\vec{x} \cap \vec{z} = \emptyset$ and $|\vec{x}| = |\vec{y}|$. Then

$$F{\bullet}C \;\stackrel{\text{def}}{=}\; (\nu\vec{z})(P\{\vec{y}/\vec{x}\} \mid Q)$$

With the help of this, our reduction system is defined almost exactly as in the monadic case:

Definition The reduction relation $\rightarrow$ over processes is the least relation satisfying the following rules:

$$\text{COMM}: \quad (\cdots + x.F) \mid (\cdots + \overline{x}.C) \rightarrow F \bullet C$$

$$\text{PAR}: \quad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad\qquad \text{RES}: \quad \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

$$\text{STRUCT}: \quad \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

■

Note that reduction is defined only over *processes*, not over arbitrary agents. Therefore in COMM, F and C must have equal arity.

The reader may have noticed that we have not defined the application Fy of an abstraction to a name. Formally there was no need to do so, because if we now define

$$((\lambda x)F)y \stackrel{\text{def}}{=} F\{y/x\}$$

then indeed every instance of application can be eliminated, using structural congruence. However, we shall freely use application. In fact if we wish to introduce (recursive) definitions of abstraction constants, such as

$$K \stackrel{\text{def}}{=} F_K$$

where F_K is an abstraction which may contain K and other abstraction constants, this application is an indispensable abbreviative device. (Recall from Section 3.1 that we use parametric recursive definition freely in examples, but ignore them in theoretical development, since they can be eliminated in favour of replication, up to weak congruence.)

5 Equivalence, Algebra and Logic

5.1 Reduction equivalence and congruence

Let us first recall from Section 2.4 the notions of *unguardness* and *observability*, and define them in the new context of polyadic π-calculus.

Definition An agent B occurs *unguarded* in A if it has some occurrence in A which is not under a prefix α. A process P is *observable at* α, written $P \downarrow_\alpha$, if some $\alpha.A$ occurs unguarded in P with α unrestricted. ■

Now we define a natural notion of bisimilarity which takes observability into account.

Definition *(Strong) reduction equivalence*, $\dot{\sim}_r$, is the largest equivalence relation $\sqsubseteq$ over processes such that $P \sqsubseteq Q$ implies

1. If $P \rightarrow P'$, then $Q \rightarrow Q'$ for some Q' such that $P' \sqsubseteq Q'$.
2. For each α, if $P \downarrow_\alpha$ then $Q \downarrow_\alpha$. ∎

This notion, also called *barbed bisimulation*, is studied by Milner and Sangiorgi [21]. Essentially a barbed equivalence is the bisimilarity induced by a reduction relation, together with an extra condition – observability. [2]

Reduction equivalence is a natural idea, but is not preserved by process constructions. For example, $x \mathrel{\dot{\sim}_r} y$ (recall that x abbreviates $x.\mathbf{0}$), but $x \,|\, \overline{x} \mathrel{\not\dot{\sim}_r} y \,|\, \overline{x}$. Since the left side has a reduction. Therefore we define

Definition *(Strong) reduction congruence*, $\sim_r$, is the largest congruence included in reduction equivalence. ∎

It is standard that $P \sim_r Q$ iff, for all process contexts $\mathcal{C}[\ ]$, $\mathcal{C}[P] \mathrel{\dot{\sim}_r} \mathcal{C}[Q]$. A process context $\mathcal{C}[\ ]$ is a process term with a single hole, such that placing a process in the hole yields a well-formed process.

The reason for imposing the observability condition in reduction equivalence is that reduction congruence then coincides exactly with the strong early congruence relation of [19]. This is proved in [21]. Here is an example, due to Gérard Boudol, showing the need for the observability condition. For this purpose we use τ, the silent (unobservable) action which we shall introduce formally in the next section. Define

$$J \stackrel{\text{def}}{=} \tau.J + a.K \quad , \quad K \stackrel{\text{def}}{=} \tau.K$$

Note that neither J nor K ever reaches a state in which a reduction is impossible. Now $J \mathrel{\not\dot{\sim}_r} K$, since $J \downarrow_a$ but $K \not\downarrow_a$. However, J and K would be congruent if $\dot{\sim}_r$ were weakened by omission of the observability condition.

Later we shall sometimes allude to *weak* reduction congruence, which is defined essentially by replacing $\rightarrow$ by its transitive reflexive closure in the above:

Definition *Weak reduction equivalence*, $\dot{\approx}_r$, is the largest equivalence relation $\sqsubseteq$ over processes such that $P \sqsubseteq Q$ implies

1. If $P \rightarrow P'$, then $Q \rightarrow^* Q'$ for some Q' such that $P' \sqsubseteq Q'$.
2. For each α, if $P \downarrow_\alpha$ then $Q \rightarrow^* \downarrow_\alpha$.

[2] It can be shown that the condition can be relaxed to simply $P\downarrow \Rightarrow Q\downarrow$, where $P\downarrow$ means $P\downarrow_\alpha$ for some α, without changing the induced congruence. But this has not been shown for the *weak* version.

Then *weak reduction congruence*, $\approx_r$, is the largest congruence included in weak reduction equivalence. $\blacksquare$

It turns out that this congruence indeed coincides with the *weak* analogue of strong early congruence.

These coincidences show that a satisfactory semantics for the π-calculus can be defined via reduction and observability. But in one sense the definitions above are unsatisfactory; quantification over all contexts is far from a direct way of characterizing a congruence, and gives little insight. We now proceed to repeat the treatment of bisimilarity in [19], though in a form more appropriate to our new presentation of the π-calculus. The bisimilarity equivalences are very close to their induced congruences.

5.2 Commitment and congruence

An atomic normal process $\alpha.A$ can be regarded as an *action* α and a *continuation* A. (It is perhaps more accurate to think of α as the *location* of an action; we have already used this term.) We shall call $\alpha.A$ a *commitment*; it is a process committed to act at α.

The idea we want to formalize is that, semantically, a process is in general nothing more than a set of commitments. (This means that every process is semantically congruent with a normal process $\Sigma\alpha_i.A_i$; we shall justify the term "set" by showing that $M + M$ is congruent with M.) The way we shall formalize it is by defining the relation

$$P \succ \alpha.A$$

between processes and commitments, pronounced "P can commit to $\alpha.A$". Of course, this is exactly what the *labelled transition system* of [19] achieved, with different notation. For example, instead of $P \succ \overline{x}.[y]P'$, the labelled transition $P \xrightarrow{\overline{x}y} P'$ was used in [19]; similarly, instead of $P \succ x.(\lambda y)P'$, the transition $P \xrightarrow{x(y)} P'$ was used. Joachim Parrow indeed suggested using $P \xrightarrow{x} (\lambda y)P'$ for the latter, and the introduction of concretions in effect allows $P \xrightarrow{\overline{x}} [y]P'$ for the former. This usage is not just notational convenience; it yields a more satisfactory presentation of π-calculus dynamics, as we see below.

Two preliminaries are necessary. First, we introduce the unobservable action τ, and henceforth we allow $\alpha, \beta, \ldots$ to stand for τ as well as for a name or co-name. [3] Second, we wish to extend composition $\mid$ to operate on abstractions and concretions (though not to compose an abstraction with a concretion). So, in line with the definition of pseudo-application in Section 3.3, let $F \equiv (\lambda\vec{x})P$ and $G \equiv (\lambda\vec{y})Q$ where the names $\vec{x}$ do not occur in G, nor $\vec{y}$ in F. Then

$$F \mid G \stackrel{\text{def}}{=} (\lambda\vec{x}\vec{y})(P \mid Q)$$

[3] In fact, the prefix τ is definable by $\tau.P \stackrel{\text{def}}{=} (\nu x)(x.P \mid \overline{x})$ where $x \notin \text{fn}(P)$.

Similarly, let $C \equiv (\nu\vec{x})[\vec{u}]P$ and $D \equiv (\nu\vec{y})[\vec{v}]Q$ where the names $\vec{x}$ do not occur in D, nor $\vec{y}$ in C. Then

$$C \mid D \stackrel{\text{def}}{=} (\nu\vec{x}\vec{y})[\vec{u}\vec{v}](P \mid Q)$$

Clearly, $\mid$ is associative up to $\equiv$, but not commutative in general (though it is so upon processes). Note also that, because a process is both an abstraction and a concretion, $A|P$ is defined for any agent A and process P; moreover, $A|P \equiv P|A$.

We are now ready to define our operational semantics in terms of commitment.

Definition The *commitment* relation $\succ$ between processes and commitments is the smallest relation satisfying the following rules:

$$\textsc{sum}: \ \cdots + \alpha.A \succ \alpha.A$$

$$\textsc{comm}: \ \frac{P \succ x.F \quad Q \succ \overline{x}.C}{P \mid Q \succ \tau.(F \bullet C)}$$

$$\textsc{par}: \ \frac{P \succ \alpha.A}{P \mid Q \succ \alpha.(A \mid Q)} \qquad \textsc{res}: \ \frac{P \succ \alpha.A}{(\nu x)P \succ \alpha.(\nu x)A} \quad (\alpha \notin \{x, \overline{x}\})$$

$$\textsc{struct}: \ \frac{Q \equiv P \quad P \succ \alpha.A \quad A \equiv B}{Q \succ \alpha.B} \qquad \blacksquare$$

The reader who is familiar with [19] will notice how much simpler our operational semantics has become. Of course, some of the complexity is concealed in the laws of structural congruence; but those laws are so to speak digestible without concern for the dynamics of action, and therefore deserve to be factored apart from the dynamics. The treatment of restriction derives further benefit from the admission of *restricted* concretions; the restriction rule RES here covers the two rules RES and OPEN of [19]. Moreover, the only remaining side condition, which is upon RES, is the *essence* of restriction; all other side-conditions in the rules of [19] were nothing more than administrative – avoiding clashes of free and bound names.

We shall proceed to define the most natural form of bisimilarity in terms of commitment. First, a desirable property of relations will make the job simpler:

Definition Let $\sqsubseteq$ be an arbitrary binary relation over agents. We say $\sqsubseteq$ is *respectable* if it includes structural congruence ($\equiv$), and moreover it is respected by decomposition of concretions and application of abstractions, i.e.

1. If $C \sqsubseteq D$ then they have standard forms $C \equiv (\nu\vec{x})[\vec{y}]P$ and $D \equiv (\nu\vec{x})[\vec{y}]Q$ such that $P \sqsubseteq Q$.

2. If $F \sqsubseteq G$ then their arities are equal, n say, and for any $\vec{y}$ of length n, $F\vec{y} \sqsubseteq G\vec{y}$. $\blacksquare$

Note that this is dual to a congruence condition; the relation is to be preserved by decomposition rather than composition.

Now we define bisimulation and bisimilarity for *all* agents, not only processes, as follows:

Definition A relation $\sqsubseteq$ over agents is a (*strong*) *simulation* if it is respectable, and also if $P \sqsubseteq Q$ and $P \succ \alpha.A$, then $Q \succ \alpha.B$ for some B such that $A \sqsubseteq B$.

$\sqsubseteq$ is a (*strong*) *bisimulation* if both $\sqsubseteq$ and its converse are simulations. (*Strong*) *bisimilarity*, $\dot\sim$, is the largest bisimulation. ■

We may also describe $\dot\sim$ as the largest respectable equivalence closed under commitment. It is the union of all bisimulations; hence to prove $P \dot\sim Q$ one need only exhibit a bisimulation containing the pair (P, Q). It is the strong *late* bisimilarity of [19].

As pointed out there, it is not quite a congruence relation. In fact, it is not preserved by substitution (of names for names); for example

$$\overline{x} \mid y \dot\sim \overline{x}.y + y.\overline{x} \qquad \text{but} \qquad \overline{x} \mid x \not\dot\sim \overline{x}.x + x.\overline{x}$$

However $\dot\sim$ is preserved by every agent construction except abstraction, (λx). It is therefore much closer to its induced congruence than is the case for reduction equivalence. To close the gap we need only impose closure under substitutions. Let σ range over substitutions, i.e. replacements $\{\vec{y}/\vec{x}\}$ of names for (distinct) names. Then:

Definition P and Q are *strongly congruent*, written $P \sim Q$, if $P\sigma \dot\sim Q\sigma$ for all substitutions σ. ■

Proposition $\sim$ is a congruence.

Proof Along the lines in [19]. ■

5.3 Axiomatization

Now following [19], but making minor adjustments to allow for abstractions and concretions, we can present an axiomatization of $\dot\sim$ which is complete for *finite* agents, i.e. those without replication.

Definition The theory SGE (Strong Ground Equivalence) is the smallest set of equations $A = B$ over agents satisfying the following (we write SGE $\vdash A = B$ to mean that $A = B \in$ SGE):

1. If $A_1 \equiv A_2$ then SGE $\vdash A_1 = A_2$

2. SGE is closed under every agent construction except abstraction. For example, if SGE $\vdash M_1 = M_2$ then SGE $\vdash M_1 + N = M_2 + N$.

3. If SGE $\vdash Fy = Gy$ for every[4] y then SGE $\vdash F = G$.

4. SGE $\vdash M + M = M$

5. SGE $\vdash (\nu x)\Sigma\alpha_i.A_i = \Sigma\alpha_i.(\nu x)A_i$, if no α_i is either x or $\overline{x}$.

6. (Expansion)

$$\begin{array}{rcl} \text{SGE} \vdash M \mid N & = & \Sigma\{\alpha.(A \mid N) : \alpha.A \text{ a summand of } M\} \\ & + & \Sigma\{\beta.(B \mid M) : \beta.B \text{ a summand of } N\} \\ & + & \Sigma\{\tau.(F \bullet C) : x.F \text{ a summand of } M \text{ (resp. } N) \\ & & \quad \text{and } \overline{x}.C \text{ a summand of } N \text{ (resp. } M)\} \end{array}$$

■

"Ground equivalence" is a synonym for "bisimilarity"; the term "ground" indicates that the theory is not closed under substitution for names.

Theorem (Soundness of SGE) If SGE $\vdash A = B$ then $A \stackrel{\cdot}{\sim} B$.

Proof Along the lines in [19]. ■

Theorem (Completeness of SGE) If A and B are finite and $A \stackrel{\cdot}{\sim} B$, then SGE $\vdash A = B$.

Proof Along the lines in [19]. ■

The essence of SGE is that two agents are equivalent iff they have equivalent commitments. The proof of completeness depends upon showing that for any P, there is a normal process (i.e. a sum of commitments) M such that SGE $\vdash P = M$.

This characterization allows us to show exactly why strong congruence, $\sim$, is in fact stronger than strong reduction congruence, $\sim_r$. For there are processes P and Q which do *not* have equivalent commitments, and yet $P \sim_r Q$. In particular, let

$$M \equiv x(u).P_1 + x(u).P_2 \quad , \quad N \equiv M + x(u).P_3$$

where P_1, P_2 and P_3 are distinct under $\sim$, but P_3 behaves like P_1 if u takes a particular value y, and otherwise behaves like P_2.[5] Then it turns out that indeed

[4] This rule is in effect finitary, since the hypothesis need only be proved for every name y free in F or G, and one new y.

[5] For example $P_1 \equiv u.\overline{y} + \overline{y}.u$, $P_2 \equiv u.\overline{y} + \overline{y}.u + \tau$, $P_3 \equiv u \mid \overline{y}$.

$M \sim_r N$, while $M \not\sim N$ since N has a commitment distinct from any commitment of M. (We omit full details of this argument.)

So how must we modify strong bisimilarity $\dot\sim$, so that its induced congruence coincides exactly with $\sim_r$? The answer is that we must relax the condition on positive commitments only.

Definition *Strong early bisimilarity*, $\dot\sim_e$, is the largest respectable equivalence $\sqsubseteq$ such that if $P \sqsubseteq Q$ then

1. If $P \succ x.F$ and F has arity n, then for each $\vec{y}$ of length n there exists G such that $Q \succ x.G$ and $F\vec{y} \sqsubseteq G\vec{y}$.
2. If $P \succ \overline{x}.C$, then $Q \succ \overline{x}.D$ for some D such that $C \sqsubseteq D$. ■

Here the condition on positive commitments is weaker, because it has $\forall\vec{y}\,\exists G$ where $\dot\sim$ effectively demands the stronger condition $\exists G\,\forall\vec{y}$. Thus, it is clear that $\dot\sim \subseteq \dot\sim_e$. Again, $\dot\sim_e$ is nearly a congruence, being closed under every construction except abstraction. So we define

Definition P and Q are *strongly early-congruent*, written $P \sim_e Q$, if $P\sigma \dot\sim_e Q\sigma$ for all substitutions σ. ■

Proposition $\sim_e$ is a congruence. ■

And finally, we have recovered reduction congruence:

Theorem (Sangiorgi) Strong early congruence coincides with reduction congruence; i.e. $\sim_e = \sim_r$. ■

We shall not consider equational laws for $\dot\sim_e$. Joachim Parrow has given an axiomatization; it involves an extra process construction which we are not using in this paper.

5.4 Properties of replication

Interesting process systems usually involve infinite behaviour, hence replication. The equational theory SGE cannot hope to prove all true equations about infinite systems – in fact, they are not recursively enumerable. All process algebras [4, 10, 13, 16] use techniques beyond purely algebraic reasoning. Here we shall use the technique of bisimulation due to Park [24]. We wish to prove three simple but important properties of replication, which will be needed later.

Proposition $!P \mid !P \sim !P$.

Proof It can be shown in a routine way that there is a bisimulation $\sqsubseteq$ whose process part (i.e. $\sqsubseteq \cap \mathcal{P}\times\mathcal{P}$) consists of all pairs

$$(\nu\vec{y})(\,!P\mid !P\mid Q)\ ,\ (\nu\vec{y})(\,!P\mid Q)$$

for any $\vec{y}$, P and Q. By taking $\vec{y}$ to be empty and $Q\equiv\mathbf{0}$, this ensures $!P\mid !P \mathrel{\dot\sim} !P$ for any P. But further, the above set of pairs is closed under substitutions, so the congruence $\sim$ also holds. ∎

This property shows that the duplication of a replicable resource has no behavioural effect, which is not surprising.

We shall now look at a more subtle property, concerning what may be called *private* resources. If a system S contains the subsystem

$$(\nu x)(P\mid !x.F)$$

then we may call $!x.F$ a *private resource* of P, because only P can acquire a replica of it. (Of course F may contain other free names, so the replica – once active – may interact with the rest of S.)

Now suppose $P\equiv P_1|P_2$ in the above. Then P_1 and P_2 share the private resource. Does it make any difference if we give each of P_1 and P_2 its own private resource? That is, is it true that

$$(\nu x)(P_1\mid P_2\mid !x.F)\ \sim\ (\nu x)(P_1\mid !x.F)\mid(\nu x)(P_2\mid !x.F)\qquad ?$$

A moment's thought reveals that this cannot hold in general. Take $P_1\equiv\overline{x}.[y]$, $P_2\equiv x.(\lambda z)\mathbf{0}$. Then not only can P_1 access the resource; it can also – on the left-hand side but not on the right-hand side – interact with P_2. Thus the bisimilarity fails. But this is only because P_2 *bears the name* x, in the terms of Section 3.4. So let us impose the condition that none of P_1, P_2 or F bears the name x. On this occasion, we shall use a slightly different extra condition from Section 3.4 to make this property invariant under action. The extra condition amounts to saying that x is only used in P_1, P_2 or F to *access* the resource; that is, it must not occur free as an *object*. Then indeed our desired result follows. To be precise:

Proposition Assume that every free occurrence of x in P_1, P_2 and F is as a negative subject. Then

$$(\nu x)(P_1\mid P_2\mid !x.F)\ \sim\ (\nu x)(P_1\mid !x.F)\mid(\nu x)(P_2\mid !x.F)$$

Proof It can be shown that there is a bisimulation $\sqsubseteq$ such that $\sqsubseteq\cap\mathcal{P}\times\mathcal{P}$ consists of all pairs

$$(\nu\vec{y})(\nu x)(P_1\mid P_2\mid !x.F)\ ,\ (\nu\vec{y})\Big((\nu x)(P_1\mid !x.F)\mid(\nu x)(P_2\mid !x.F)\Big)$$

for any P_1, P_2, x, F and $\vec{y}$ such that x occurs free in P_1, P_2 and F only as a negative subject. It can be checked that this relation is closed under substitutions, and hence the result follows by taking $\vec{y}$ to be empty. ■

As we shall see in Section 6.3, persistent data structures are an instance of replicable resources. So this proposition can be interpreted as saying that it makes no difference whether two processes share a data structure, or each has its own private copy.

A good way to think of the previous proposition is "a private resource can be distributed over composition". This immediately suggests the question " ... and over what else?" Obviously we hope it can be distributed over replication, and this is indeed true.

Proposition Assume that every free occurrence of x in P and F is as a negative subject. Then

$$(\nu x)(\,!P \mid !x.F) \;\sim\; !(\nu x)(P \mid !x.F)$$

Proof We proceed much as before, but using the notion of bisimulation *up to* $\dot{\sim}$ from [16]. This just means that we can use known bisimilarities when exhibiting new bisimulations. Using the previous proposition, it can be shown that there is a bisimulation up to $\dot{\sim}$ containing the process-pairs

$$(\nu\vec{y})(\nu x)(\,!P \mid !x.F \mid Q)\;,\;(\nu\vec{y})\Big(\,!(\nu x)(P \mid !x.F) \mid (\nu x)(Q \mid !x.F)\Big)$$

for any P, Q, x, F and $\vec{y}$ such that x occurs free in P, Q and F only as a negative subject. Then we take $\vec{y}$ empty and $Q \equiv \mathbf{0}$ to get the result. ■

A striking consequence of these two propositions, as we shall see in Section 6.4, is that β-conversion is equationally valid in the interpretation of λ-calculus in π-calculus. Essentially, this is because we model *application* of an abstraction $\lambda x M$ to a term N in λ-calculus by providing M with access – via x – to the resource N.

It therefore appears that these properties of replication have quite wide applicability, since computational phenomena which appear significantly different can be seen as accessing resources.

5.5 Logical characterization

In [20], a modal logic was defined to give an alternative characterization of the bisimilarity relations in π-calculus, following a familiar line in process algebra. It was first done for CCS by Hennessy and Milner [11]; see also Milner [16], Chapter 10. No inference system was defined for this logic; the aim was just to define the *satisfaction* relation $P \models \varphi$ between processes P and logical formulae φ, in such a way that P and Q are bisimilar iff they satisfy exactly the same formulae.

The main attention in [20] was upon the modalities, and in particular the modality for input:

$$\langle x(y) \rangle \varphi$$

Because "late" and "early" strong bisimilarity differ just in their requirement upon input transitions $P \stackrel{x(y)}{\to} P'$, the input modality must have two versions:[6]

$$P \models \langle x(y) \rangle^l \varphi \quad \text{iff} \quad \exists P' \, \forall z \,.\, P \stackrel{x(y)}{\to} P' \text{ and } P'\{z/y\} \models \varphi\{z/y\}$$
$$P \models \langle x(y) \rangle^e \varphi \quad \text{iff} \quad \forall z \, \exists P' \,.\, P \stackrel{x(y)}{\to} P' \text{ and } P'\{z/y\} \models \varphi\{z/y\}$$

where we have highlighted the only difference – quantifier inversion.

Now, we are representing $P \stackrel{x(y)}{\to} P'$ as the commitment $P \succ x.(\lambda y)P'$; this factoring of input prefix into two parts, location and abstraction, allows us to simplify our logic by a similar factoring. This holds for output modalities too. In fact, we find that the logical constructions for abstraction and concretion are, as one might hope, dependent product and dependent sum; also, the action modality becomes suitably primitive.

Our logic $\mathcal{L}$, which will characterize late bisimilarity, is the set of formulae φ given by the syntax[7]

$$\varphi \ ::= \ \top \ \mid \ \varphi \wedge \psi \ \mid \ \neg\varphi \ \mid \ x = y \ \mid \ \langle \alpha \rangle \varphi \ \mid \ (\Sigma x)\varphi \ \mid \ (\Pi x)\varphi$$

(where α ranges, as before, over names, co-names and τ). The last two, sum and product, bind x; they will only be satisfied respectively by concretions and abstractions with non-zero arity. On the other hand $\langle \alpha \rangle \varphi$ will only be satisfied by processes.

Definition The *satisfaction* relation $\models$ between agents and formulae is given by induction on formula size, as follows:

$A \models \top$		always
$A \models \varphi \wedge \psi$	iff	$A \models \varphi$ and $A \models \psi$
$A \models \neg\varphi$	iff	not $A \models \varphi$
$A \models x = y$	iff	x and y are the same name
$A \models \langle \alpha \rangle \varphi$	iff	for some A', $A \succ \alpha.A'$ and $A' \models \varphi$
$A \models (\Sigma x)\varphi$	iff	$A \equiv [y]C$ or $(\nu y)[y]C$, with $y \notin \mathrm{fn}((\Sigma x)\varphi)$ in the latter case, and $C \models \varphi\{y/x\}$
$A \models (\Pi x)\varphi$	iff	for all y, $Ay \models \varphi\{y/x\}$ ∎

Note that $\langle x \rangle (\Pi y)\varphi$ is exactly $\langle x(y) \rangle^l$; we shall consider $\langle x(y) \rangle^e$ later. There is another intriguing point; one might have expected to need restriction (νy) in the

[6] To be exact, in these definition we require $y \notin \mathrm{fn}(P)$.

[7] In [20], the form $[x = y]\varphi$ (meaning "if $x = y$ then φ") was used in place of the atomic formula $x = y$. But the two are interdefinable in the presence of $\neg$ and $\wedge$.

logic, to cope with the output modality – more exactly the bound data-name in a concretion. But the side condition on y in the $(\Sigma x)\varphi$ case takes care of restrictions in agents. To clarify this, consider $P \equiv \overline{x}.[y]$ and $Q \equiv \overline{x}.(\nu y)[y]$; they are not bisimilar, so – in view of the characterization theorem below – there must be a formula which distinguishes them. In fact, take $\varphi \equiv \langle\overline{x}\rangle(\Sigma z)(z = y)$; then indeed $P \models \varphi$, $Q \not\models \varphi$.

Now the proof of the following can be done along the same lines as in [20]:

Theorem (logical characterisation of $\dot\sim$) $A \dot\sim B$ iff for every $\varphi \in \mathcal{L}$,

$$A \models \varphi \text{ iff } B \models \varphi \qquad \blacksquare$$

Our next task is to see how to weaken $\mathcal{L}$, in order to achieve a logic which characterizes the weaker (i.e. larger) equivalence $\dot\sim_r$, reduction equivalence or early bisimilarity. The key is that, for $P \dot\sim_r Q$, we do not demand that every abstraction F, to which P can commit at x, must be matched by Q with such an abstraction G; the equivalence only depends on matching every *process instance* of such an abstraction, i.e. every pair $\vec{y}, P'$ such that for some F, $P \succ x.F$ and $F\vec{y}\equiv P'$. Thus the logic must be weakened so that $P \models \varphi$ cannot depend directly on properties of each F for which $P \succ x.F$. This entails removing from $\mathcal{L}$ the positive action modalities $\langle x\rangle$, and replacing them with – in effect – the polyadic version of $\langle x(y)\rangle^e\varphi$, namely:

$$P \models \langle x(\vec{y})\rangle^e\varphi \quad \text{iff} \quad \text{for all } \vec{z},\ P \succ (\lambda\vec{y})P' \text{ for some } P' \\ \text{such that } P'\{\vec{z}/\vec{y}\} \models \varphi\{\vec{z}/\vec{y}\}$$

Let us call this weakened logic $\mathcal{L}^r$. Then indeed,

Theorem (logical characterization of $\dot\sim_r$) $A \dot\sim_r B$ iff, for every $\varphi \in \mathcal{L}^r$,

$$A \models \varphi \text{ iff } B \models \varphi \qquad \blacksquare$$

6 Sorts, Data structures and Functions

If we look at the examples which we have used hitherto to illustrate the π-calculus, we see that each one obeys some discipline in its use of names. By this, we mean something very simple indeed: just the length and nature of the vector of names which a given name may carry in a communication. For the numerals of Section 3.3, all names carry the empty vector. For the mobile phones of Section 3.2 it is more interesting; *alert*, *give* and *talk* all carry the empty vector, but *switch* carries a pair. This is not just any pair; it is a $(talk, switch)$ pair. For the truth-values of Section 4.2, t and f carry nothing, but a boolean location like b carries a (t, f) pair.

It may be that *any* realistic application of the π-calculus is disciplined in a natural way, but the discipline can be different in each case. A loose analogy is that when the (untyped) λ-calculus is used in an application, rather than studied in its own right, there is almost always a type discipline of some kind; e.g. the simple type hierarchy, or the second-order λ-calculus, or a system of value-dependent types.

The kind of name-use discipline which first comes to mind, for the π-calculus, would employ something like the *arities* of Martin-Löf; an arity in this sense is just a properly nested sequence of parentheses. A name which carries nothing would have arity (); a name which carries a vector of n names with arities $a_1, \ldots, a_n$ would have arity $(a_1 \cdots a_n)$. But this is *too* simple! Such a hierarchy of arities does not work, because a name must sometimes carry another name "of the same kind" – i.e. of the same arity – as itself; witness *switch* in the mobile phone example. We now propose a discipline of *sorts* which is as simple as possible, while admitting this kind of circularity (which amounts to admitting a kind of self-reference).

6.1 Sorts and sortings

Assume now a basic collection $\mathcal{S}$ of *subject sorts* and for each $S \in \mathcal{S}$ an infinity of names with subject sort S (write $x : S$). Then the *object sorts* $Ob(\mathcal{S})$ are just sequences over $\mathcal{S}$; that is

$$Ob(\mathcal{S}) \quad = \quad \mathcal{S}^*$$

We shall write $(S_1 \cdots S_n)$, possibly interspersed with commas, for an object sort; the empty object sort is (). We let $s, t, \ldots$ range over object sorts. We use $s\widehat{\ }t$ for the concatenation of object sorts; e.g. $(S_1)\widehat{\ }(S_2S_3) = (S_1S_2S_3)$.

Now we define a *sorting* over $\mathcal{S}$ to be a non-empty partial function

$$ob \quad : \quad \mathcal{S} \rightharpoonup Ob(\mathcal{S})$$

If ob is finite, we typically write it as $\{\, S_1 \mapsto ob(S_1), \ldots, S_n \mapsto ob(S_n) \,\}$. A sorting just describes, for any name $x : S$, the sort of name-vector which it can carry. Thus, for the numerals of Section 3.3 we have the uninteresting sorting

$$\{\, \text{SUCC} \mapsto (),\ \text{ZERO} \mapsto () \,\}$$

with $x, y : \text{SUCC}$ and $z, w : \text{ZERO}$. For the phones of Section 3.2, it is a little more interesting:

$$\{\, \text{ALERT} \mapsto (),\ \text{GIVE} \mapsto (),\ \text{TALK} \mapsto (),\ \text{SWITCH} \mapsto (\text{TALK}, \text{SWITCH}) \,\}$$

with $alert_i : \text{ALERT}, \ldots,$ and $switch_i : \text{SWITCH}$. Note that there is little reason to distinguish ALERT from GIVE; but we should distinguish TALK, since the distinction gives more precise information about the kind of messages which can be carried on a SWITCH channel.

Given a sorting ob, we must give the conditions under which an agent is said to *respect ob*. To this end, we show how to ascribe an object sort to each suitable agent, equal in length to its numeric arity; thus a process always has sort ().

Definition An agent A *respects* a sorting ob, or is *well-sorted* for ob, if we can infer $A : s$ for some object sort s from the following *formation rules*:

$$\frac{x : S \qquad F : ob(S)}{x.F : ()} \qquad\qquad \frac{x : S \qquad C : ob(S)}{\overline{x}.C : ()} \qquad\qquad \frac{P : ()}{\tau.P : ()}$$

$$\mathbf{0} : () \qquad\qquad \frac{M : () \qquad N : ()}{M + N : ()}$$

$$\frac{P : () \qquad Q : ()}{P \mid Q : ()} \qquad\qquad \frac{P : ()}{!P : ()} \qquad\qquad \frac{A : s}{(\nu x)A : s}$$

$$\frac{x : S \qquad F : s}{(\lambda x)F : (S)\hat{\ }s} \qquad\qquad \frac{x : S \qquad C : s}{[x]C : (S)\hat{\ }s}$$

■

Exercise First prove – or assume – that if x and y have equal sort and $A : s$, then $A\{y/x\} : s$. Next prove – or assume – that if $A : s$ and $A \equiv B$, then $B : s$. (Assume that in a change of bound names, a name is replaced only by another of equal sort.)

Now recall the definition of application and composition of abstractions, in Sections 4.3 and 5.2. Prove that the following formation rules are admissible:

$$\frac{F : (S)\hat{\ }s \qquad y : S}{Fy : s} \qquad\qquad \frac{F : s \qquad G : t}{F \mid G : s\hat{\ }t}$$

(A rule is *admissible* if every proof using the rule can be transformed into one which does not use it.) ■

Some simple sortings correspond to familiar calculi. The simplest sorting of all, $\{\,\text{NAME} \mapsto ()\,\}$ – one subject sort carrying nothing – is just CCS; the next simplest, $\{\,\text{NAME} \mapsto (\text{NAME})\,\}$, is just the monadic π-calculus.

Of course there are more refined sortings for the monadic π-calculus; they will classify the use of names, but clearly $ob(S)$ will always be a singleton sequence for any S. Recall the encoding of multiple inputs and outputs into monadic π-calculus given in Section 3.1; for example, in the notation used there,

$$x(y_1 \cdots y_n).P \quad\mapsto\quad x(w).w(y_1).\cdots.w(y_n).P$$

This translation *destroys* well-sortedness! For if $y_1, \ldots, y_n$ have different sorts then, whatever subject sort we choose for w, the right-hand side will be ill-sorted. This shows that polyadicity admits a sort discipline which was not possible in the monadic π-calculus. This is the second reason for introducing polyadicity, which we promised at the beginning of Chapter 4.

6.2 Data structures

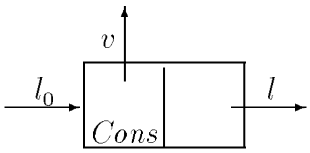
The representation of natural numbers in Section 3.3 was rather rough and ready, and does not generalize to arbitrary data structures (by which we mean data freely constructed using finitely many constructors). Let us illustrate a general method by defining single-level list structures, over elements represented by a subject sort VAL. The sorting will be

$$\{\,\text{LIST} \mapsto (\text{CONS}, \text{NIL}),\ \text{CONS} \mapsto (\text{VAL}, \text{LIST}),\ \text{NIL} \mapsto (\,)\,\}$$

and the constructors are *Cons*, *Nil* given by

$$\begin{aligned} \mathit{Cons}(v, l) &\stackrel{\text{def}}{=} (\lambda cn)\overline{c}.[vl] \\ \mathit{Nil} &\stackrel{\text{def}}{=} (\lambda cn)\overline{n} \end{aligned}$$

where $\mathit{Cons} : (\text{VAL}, \text{LIST})\hat{\ }(\text{CONS}, \text{NIL})$ and $\mathit{Nil} : (\text{CONS}, \text{NIL})$. We can think of such simple abstractions – $\mathit{Cons}(v, l)$ and *Nil* – as *nodes* of a data structure; in particular *Nil* is a *leaf* node. They are *unlocated*; but such a node – in this case a list node – can be *located* by a name of sort LIST. Thus a *Cons* value located at l_0 is $l_0.\mathit{Cons}(v, l)$, and corresponds to the familiar picture of a list cell:

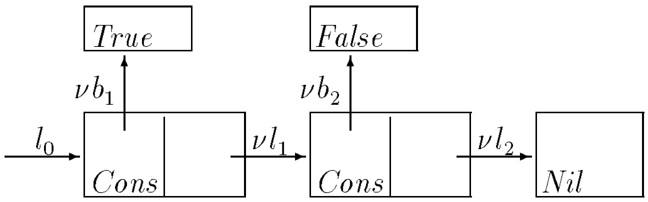


One can think of this located node as follows: at its "address" l_0 you send it a "form" with two sections, one of which must be filled in. If the value is a *Cons* it fills in the first section, c, with its components and signs it; if it is a *Nil* it signs the second section, n (there is nothing more to fill in).

Now let us consider lists of truth-values, setting VAL equal to BOOL. What is the *complete* list containing (say) the two truth-values *True* and *False*? As a restricted composition of list nodes and truth-values, it is

$$\begin{aligned} L(l_0) \stackrel{\text{def}}{=}\ & (\nu b_1 l_1)\big(l_0.\mathit{Cons}(b_1, l_1) \mid b_1.\mathit{True} \mid \\ & (\nu b_2 l_2)(l_1.\mathit{Cons}(b_2, l_2) \mid b_2.\mathit{False} \mid l_2.\mathit{Nil})\big) \end{aligned} \tag{1}$$

Note that l_0 is the only free name. Here is the diagram, using ν to mark private locations:



Note that this diagram, besides being the standard way of picturing linked lists, is actually a flow graph drawn in the usual manner for process algebra.

Exercise Revisit the numerals of Section 3.3. Now give a representation of natural numbers analogous to the above for lists, in terms of the sorting

$$\{\, \text{NAT} \mapsto (\text{SUCC}, \text{ZERO}),\ \text{SUCC} \mapsto (\text{NAT}),\ \text{ZERO} \mapsto ()\,\} \qquad \blacksquare$$

At this point, the general pattern for data structures should be clear; also, clearly the truth-values of Section 4.2 follow the pattern. By analogy with the case-analysis on truth-values, defined earlier, we can give a concretion for case-analysis on lists:

$$\mathit{Listcases}(F, Q) \ \stackrel{\text{def}}{=}\ (\nu cn)[cn](c.F + n.Q)$$

Now to do a little programming on lists, let us first define a sugared form of the co-located case-analysis $\overline{l_0}.\,\mathit{Listcases}((\lambda\, vl)P, Q)$, in the style of Standard ML, as follows:

$$\begin{array}{ll} \texttt{case}\ l_0\ \texttt{of} & : \mathit{Cons}(v, l) \Rightarrow P \\ & : \mathit{Nil} \Rightarrow Q \end{array}$$

(Note that the constructions between ":" and $\Rightarrow$ are patterns, binding the variables v and l in P.) Now define the *Append* function to concatenate lists, in the same way that we defined addition on numerals:

$$\begin{array}{rcl} \mathit{Copy}(l, m) & \stackrel{\text{def}}{=} & \texttt{case}\ l\ \texttt{of} \\ & & : \mathit{Cons}(v, l') \Rightarrow (\nu m')\big(m.\mathit{Cons}(v, m') \mid \mathit{Copy}(l', m')\big) \\ & & : \mathit{Nil} \Rightarrow m.\mathit{Nil} \\ \mathit{Append}(k, l, m) & \stackrel{\text{def}}{=} & \texttt{case}\ k\ \texttt{of} \\ & & : \mathit{Cons}(v, k') \Rightarrow (\nu m')\big(m.\mathit{Cons}(v, m') \mid \mathit{Append}(k', l, m')\big) \\ & & : \mathit{Nil} \Rightarrow \mathit{Copy}(l, m) \end{array}$$

Then, if $K(k)$ and $L(l)$ are expressions like (1) representing two lists, and if $M(m)$ is an expression representing the concatenation of these two lists, we shall indeed have

$$(\nu lm)(K(k) \mid L(l) \mid \mathit{Append}(k, l, m)) \ \approx M(m)$$

The expression (1) exhibits how a list is built from located values and located nodes. Notice that $L(l_0)$ is a *located* list; its location is l_0, the location of its root node. Interestingly enough there is no subexpression of (1) which corresponds either to the *unlocated* list containing *True* and *False*, or to the unlocated sublist containing just *False*. But we can transform $L(l_0)$ to a strongly bisimilar form which does contain such subexpressions. In fact

$$L(l_0) \sim l_0.L_0$$

where L_0, L_1 and L_2 are the unlocated list-values given by

$$\begin{aligned}
L_0 &\stackrel{\text{def}}{=} (\lambda cn)\overline{c}.(\nu bl)[bl](b.\mathit{True} \mid l.L_1) \\
L_1 &\stackrel{\text{def}}{=} (\lambda cn)\overline{c}.(\nu bl)[bl](b.\mathit{False} \mid l.L_2) \\
L_2 &\stackrel{\text{def}}{=} \mathit{Nil} \stackrel{\text{def}}{=} (\lambda cn)\overline{n}
\end{aligned} \tag{2}$$

Notice that these are closed expressions, just as *True* and *False* are closed. So they mean the same wherever they are used; it is therefore reasonable to refer to such terms as *values*.

6.3 Persistent values

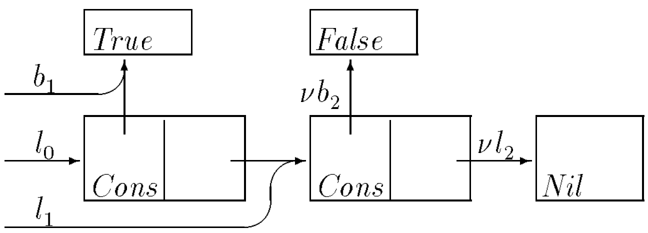
In the above treatment, the data structures are purely ephemeral; accessing them destroys them. But by use of replication they can be made persistent. Reverting to $L(l_0)$ at (1), the natural thing to do is to replicate the nodes and the component values, giving

$$\begin{aligned}
M_0(l_0) &\stackrel{\text{def}}{=} (\nu b_1 l_1)\Big(!l_0.\mathit{Cons}(b_1, l_1) \mid !b_1.\mathit{True} \mid M_1(l_1)\Big) \\
M_1(l_1) &\stackrel{\text{def}}{=} (\nu b_2 l_2)\Big(!l_1.\mathit{Cons}(b_2, l_2) \mid !b_2.\mathit{False} \mid M_2(l_2)\Big) \\
M_2(l_2) &\stackrel{\text{def}}{=} !l_2.\mathit{Nil}
\end{aligned} \tag{3}$$

Now let us see what happens when we interrogate $M_0(l_0)$. Let C be the case-analysis concretion at (2); we get

$$C \mid M_0(l_0) \;\rightarrow^2\; (\nu b_1 l_1)\Big(P\{{}^{b_1 l_1}/_{\nu l}\} \mid !l_0.\mathit{Cons}(b_1, l_1) \mid !b_1.\mathit{True} \mid M_1(l_1)\Big) \tag{4}$$

Thus P is now seeing the following structure, along the links l_0, l_1 and b_1. Note particularly the sharing of pointers:



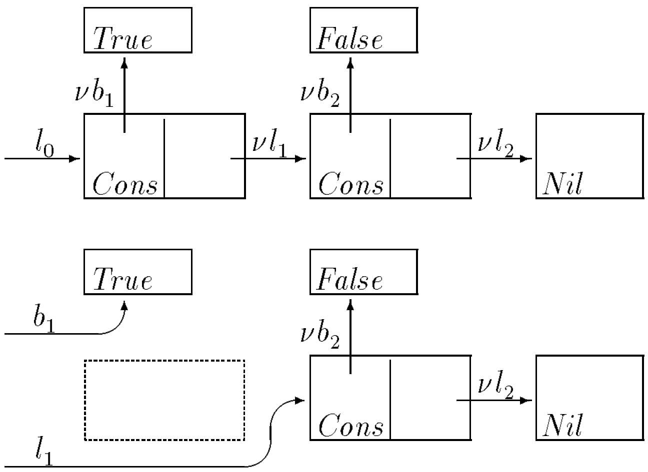
We get a different story if we apply replication, not to the *nodes* as we have just done, but to the *sublists*. This is best done on the form (2); we consider the located list $!l_0.N_0$ where

$$\begin{aligned} N_0 &\stackrel{\text{def}}{=} (\lambda cn)\overline{c}.(\nu bl)[bl](\,!b.\mathit{True}\,|\,!l.N_1) \\ N_1 &\stackrel{\text{def}}{=} (\lambda cn)\overline{c}.(\nu bl)[bl](\,!b.\mathit{False}\,|\,!l.N_2) \\ N_2 &\stackrel{\text{def}}{=} \mathit{Nil} \stackrel{\text{def}}{=} (\lambda cn)\overline{n} \end{aligned}$$

Let us interrogate $!l_0.N_0$ as we did $M_0(l_0)$. This time we get

$$C\,|\,!N_0(l_0) \;\;\rightarrow^2\;\; (\nu b_1 l_1)(P\{^{b_1 l_1}/_{vl}\}\,|\,!l_0.N_0\,|\,!b_1.\mathit{True}\,|\,!l_1.N_1)$$

Comparing with (4) we see that there is no sharing of the pointers b_1 and l_1 – because these names are not free in $!l_0.N_0$. So P is now seeing a different structure along the links l_0, l_1 and b_1:



The diagram makes it clear that, each time the complete list is traversed from l_0, a *new* copy of each component is encountered.

We have taken some care to present this phenomenon, because it is the kind of distinction which can give rise to subtle errors in programs – even sequential programs. Of course the distinction is most likely to cause serious behavioural difference when the list elements are not just values like *True* and *False*, but are instead storage cells whose stored values may be updated. The distinction is that in the second case the storage cell is copied by the replication, while in the first case only the list nodes are copied. This is reminiscent of the distinction between PASCAL's call-by-value and call-by-name parameter-passing mechanisms, in the case of scalar variables or arrays.

Burstall [7] addressed the problem of giving rigorous proofs about list-processing and tree-processing, including the use of assignment. There are some interesting features in common; he used names as locations of list segments, and carried out succinct program proofs using terms like $x \stackrel{\ell}{\rightarrow} y$, standing for a list segment starting at location x, finishing at y, containing the element-sequence ℓ, and with all internal locations distinct. It would be intriguing to encode these entities and the proofs into π-calculus using restriction.

We have seen that the π-calculus can model refined phenomena of data storage. For lack of space we shall not deal with updatable storage cells, though they are quite straightforward – following the method of defining registers in CCS, in Chapter 8 of [16].

Exercise We have seen that $M_0(l_0)$ and $!l_0.N_0$ behave differently, and indeed they are not bisimilar. Try to find a logical formula, in the logic of Section 5.5, which is satisfied by one but not the other. (*Hint*: test for sharing.) ∎

6.4 Functions

In [17] it was shown how to translate the lazy λ-calculus into π-calculus; the translation was discussed fully there, and we shall not go into great detail here. But it is worth repeating the translation in the polyadic setting – particularly because we can present the sorting which it respects.

Recall that the terms $M, N, \ldots$ of λ-calculus are given by

$$M \ ::= \ x \ \big| \ \underline{\lambda} x M \ \big| \ MN$$

where x ranges over variables. Just for this section we shall underline $\underline{\lambda}$ in the λ-calculus to distinguish λ-calculus abstractions from π-calculus abstractions.

There are many reduction relations $\rightarrow$ for the λ-calculus, many of which satisfy the rule

$$\beta : \ (\underline{\lambda} x M)N \rightarrow M\{^N\!/x\}$$

The relations differ as to which contexts admit reduction. The simplest, in some sense, is that which admits reduction only at the extreme left end of a term. This is known as lazy reduction, and its model theory has recently been investigated in detail by Abramsky [1]. Thus the *lazy reduction relation* $\rightarrow$ over λ-calculus terms is the smallest which satisfies β, together with the rule

$$\text{APPL} \ : \ \frac{M \rightarrow M'}{MN \rightarrow M'N}$$

For our translation, we introduce a subject sort VAR in the π-calculus; we take the names of sort VAR to be exactly the variables $x, y, \ldots$ of the λ-calculus. Intuitively, such a name is the location of the *argument* to a function. We also introduce a subject sort ARGS, with names $u, v, \ldots$; these names locate argument-*sequences*. Thus a term M of λ-calculus is translated into a π-calculus abstraction $(\lambda u)P$; if M reduces to a $\underline{\lambda}$-abstraction $\underline{\lambda} xN$, then correspondingly P will – after reduction – receive its argument sequence at u, and will name the first of these arguments x. In fact, an argument sequence is represented by a pair; the name x of the first argument, and the name v of the ensuing sequence. This is reflected in the sorting

$$\{\, \text{VAR} \mapsto (\text{ARGS}),\ \ \text{ARGS} \mapsto (\text{VAR}, \text{ARGS})\,\}$$

and the translation $[\![-]\!]$, given below, is easily seen to respect this sorting:

$$
\begin{aligned}
[\![\underline{\lambda} x M]\!] &\stackrel{\text{def}}{=} (\lambda u)u.(\lambda x)[\![M]\!] \\
[\![x]\!] &\stackrel{\text{def}}{=} (\lambda u)\overline{x}.[u] \\
[\![MN]\!] &\stackrel{\text{def}}{=} (\lambda u)(\nu v)\Big([\![M]\!]v \mid (\nu x)(\overline{v}.[xu] \mid !x.[\![N]\!])\Big)
\end{aligned}
$$

Note, in the third equation, how M and its argument-list are "co-located" at v. Note also the replication of N; this is because M may use its (first) argument repeatedly.

It is important to note that this translation is specific to the *lazy* reduction strategy. The theorems in [17] show that lazy reduction is closely simulated by π-calculus reduction of the translated terms. In [17] a different translation was also given for Plotkin's call-by-value λ-calculus; it is striking that the latter translation respects a different sorting.

We shall now outline the proof that β-reduction is equationally valid in our π-calculus interpretation of lazy λ-calculus. First, we prove that if M is any term of λ-calculus, then to provide $[\![M]\!]$ with a replicable resource consisting of $[\![N]\!]$ located at x is behaviourally equivalent to $[\![M\{^N\!/\!x\}]\!]$:

Lemma If x is not free in N then

$$(\nu x)([\![M]\!] \mid !x.[\![N]\!]) \;\approx\; [\![M\{^N\!/\!x\}]\!]$$

Proof First, we note that the set of equations of this form can be shown to be closed under substitutions; therefore it will be enough to prove the result with $\dot{\approx}$ in place of $\approx$. Note also that the equation is between abstractions; applying both sides to arbitrary $u{:}\text{ARGS}$, we need to show $(\nu x)([\![M]\!]u \mid !x.[\![N]\!]) \mathrel{\dot{\approx}} [\![M\{^N\!/\!x\}]\!]u$.

The proof proceeds by induction on the structure of M. We shall not give details, but only draw attention to one important step. In the case $M \equiv M_1 M_2$, the last two propositions about replication in Section 5.4 justify the creation of two copies of the private resource $!x.[\![N]\!]$, for use by M_1 and M_2 separately. All the other cases only involve a little reduction, and use of the inductive hypothesis. ■

We can now sketch a proof of the main result.

Theorem $[\![(\underline{\lambda} x M)N]\!] \approx [\![M\{^N\!/\!x\}]\!]$.

Proof We assume w.l.o.g. that x does not occur free in N. This is justified because our translation respects change of bound variables in λ-calculus. Next, for the same reasons as before, we need only demonstrate weak bisimilarity $\dot{\approx}$ to conclude the theorem.

Now by doing a single reduction we show that, for any $u{:}\text{ARGS}$,

$$\begin{array}{rcl} [\![(\underline{\lambda}xM)N]\!]u & \dot{\sim} & \tau.(\nu x)([\![M]\!]u \mid !x.[\![N]\!]) \\ & \dot{\approx} & (\nu x)([\![M]\!]u \mid !x.[\![N]\!]) \end{array}$$

and it only remains to apply the lemma. ■

7 Higher-order π-calculus

Recall from Section 6.1 that pure CCS [16], in which communication carries no data, corresponds to the sorting $\{\text{NAME} \mapsto ()\}$ in π-calculus. Without flow of data in communication, one cannot – except in a very indirect way – represent *mobility* among processes; that is, the dynamic change of neighbourhood. The π-calculus allows *names* to flow in communication, and this achieves mobility in a rather concrete way. Another approach is to allow *processes* themselves to flow in communication; a process P may send to process Q a message which consists of a third process R. Various authors have studied process flow. In particular, Thomsen [27] has developed an algebra of higher-order processes, CHOCS, based upon a natural extension of the operational semantics of CCS.

In what follows we shall describe the first component of a concretion as a *datum*. Thus effectively CHOCS allows processes as data, while π-calculus allows only names.

7.1 Processes as data

Part of the motivation of the π-calculus was that one should get all the effect of processes as data, simply by using names as data. Crudely speaking: Instead of sending you a process R, I send you a name which gives you access to R. As a simple example consider $P|Q$, where

$$P \stackrel{\text{def}}{=} \overline{x}.[R]P' \quad \text{and} \quad Q \stackrel{\text{def}}{=} x.(\lambda X)(X \mid Q')$$

This is not π-calculus as we have defined it, because the concretion has a process R as datum, and X in the abstraction is not a name but a variable over processes. But in a higher-order calculus which allows this, we would expect the reduction

$$P \mid Q \;\rightarrow\; P' \mid R \mid Q'$$

Now, we can get the same effect by *locating* R at a new name z, and sending z:

$$\begin{array}{rcl} \widehat{P} \mid \widehat{Q} & \equiv & \overline{x}.(\nu z)[z](z.R \mid P') \mid x.(\lambda z)(\overline{z} \mid Q') \\ & \rightarrow & (\nu z)\big((z.R \mid P') \mid (\overline{z} \mid Q')\big) \\ & \rightarrow & P' \mid R \mid Q' \end{array}$$

There are two issues about such an encoding. First, can it be made to work in general? (The above is a special case, and ignores some complications.) Second – and independently – what pleasant properties may be found for the higher-order calculus which are not enjoyed by the first-order π-calculus? One such property may be clarity of expression; even though the encoding may work, the encoded expressions may be obscure.

Here we address the first question: Does the encoding work? Thomsen first examined this. He gave a translation – which we shall write $(\widehat{\ })$ – from Plain CHOCS into π-calculus; then he exhibited a detailed correspondence between the operational behaviours of P and $\widehat{P}$. Ideally, one would like to prove the double implication

$$P \simeq Q \iff \widehat{P} \simeq \widehat{Q}$$

for some natural congruence $\simeq$. Thomsen came close to proving this for $\approx$, observation congruence; but unfortunately the double implication appears to fail in both directions, for subtle reasons. However, stimulated by Thomsen's work, Sangiorgi has been able to show that the implication does indeed hold in both directions, when $\simeq$ is taken as *weak reduction congruence* $\approx_r$; this is a natural analogue of the strong reduction congruence $\sim_r$ introduced in Section 5.1. Furthermore the results holds not only for processes as data, but also – under a sorting constraint – when data may be *process abstractions* of arbitrary high order.

7.2 Syntax and commitment rules

To extend our syntax to higher order we must first decide what data to allow. We could admit just processes as data, as in CHOCS. But we prefer to go further and admit *parametrized* processes, i.e. abstractions, as data. This adds considerable expressive power even if we only admit processes with *name parameters*; for example, we can define the chaining combinator of Section 4.1 as an abstraction, which we could not do if only processes were admitted as data, as follows:

$$\frown \ \stackrel{\text{def}}{=}\ (\lambda XY)(\lambda xy)(\nu z)(Xxz \mid Yzy)$$

Here, X and Y stand for processes abstracted upon two names; one can see how this parameterization does –among other things – the job of the renaming operator of CCS.

But we can go further and allow the parameters themselves to be abstractions; thus we may proceed directly from first-order π-calculus (with just names as data) to ω-order π-calculus.

The change needed from the syntax of Section 4.3 is slight. Here we give the unsorted version; we treat higher-order sorts in Section 7.3 below. First we must introduce *abstraction variables* $X, Y, \ldots$; then the syntax of abstractions becomes

$$\begin{array}{llll} \text{Abstractions :} & F & ::= & P \mid (\lambda x)F \mid (\lambda X)F \mid (\nu x)F \\ & & & \mid X \mid Fx \mid FG \end{array}$$

Notice that application is now introduced explicitly; this is needed because abstraction variables are now present (see the example above). Next, concretions may now contain abstractions as data, so they become

$$\text{Concretions :} \qquad C \ ::= \ P \ \mid \ [x]C \ \mid \ [F]C \ \mid \ (\nu x)C$$

Finally, processes remain unchanged except that we must admit an abstraction expression (e.g. Xxz in the example) as a process, so we add a clause:

$$\text{Processes :} \qquad P \ ::= \ \cdots \ \mid \ F$$

The structural congruence rules of Section 4.3 only need obvious extensions; for example we add the following to rule 8 (for concretions):

$$(\nu y)[F]C \equiv [F](\nu y)C \qquad (y \notin \mathrm{fn}(F))$$

We leave it to the reader to supply the obvious rule for the application FG of one abstraction to another, and the obvious extension to the definition of pseudo-application, $F \bullet C$.

Then no change at all is needed to the reduction rules (Section 4.3), nor to the definitions of observability $\downarrow_\alpha$ (Section 2.4), reduction equivalence $\dot{\sim}_r$ and congruence $\sim_r$(Section 5.1), and commitment $\succ$ (Section 5.2). (A minor point: It is natural to confine these definitions to agents with no free abstraction variables.)

By stark constrast, a subtle and difficult question arises in trying to generalize the definition of strong bisimilarity $\dot{\sim}$ in Section 5.2. We shall content ourselves here with an intuitive description of the problem. The issue is to do with concretions, and it is this: If $P \succ \overline{z}.C$, then for $P \dot{\sim} Q$ to hold we must require that $Q \succ \overline{z}.D$ for some concretion D which corresponds suitably to C. What should the correspondence be? In Section 5.2 we demanded that they have identical data; that is, $C \equiv (\nu\vec{x})[\vec{y}]P'$ and $D \equiv (\nu\vec{x})[\vec{y}]Q'$ where $P' \dot{\sim} Q'$. This is appropriate for data *names*, but not for data *abstractions*. In the latter case, it is more appropriate to ask that they be *bisimilar*, not identical. Indeed, Thomsen took this course.

This is a plausible requirement, but one can argue that it is too strong. Consider for example $C \equiv (\nu x)[\overline{x}.\mathbf{0}]P'$ and $D \equiv (\nu x)[\mathbf{0}]Q'$. Certainly $\overline{x}.\mathbf{0} \not\dot{\sim} \mathbf{0}$; but because $\overline{x}.\mathbf{0}$ is in some sense "bound to P'" by the restriction (νx) , its $\overline{x}$-action cannot be observed directly. Moreover if we now take $P' \equiv Q' \equiv \mathbf{0}$ then there is no way in which this action can ever be complemented, and in this case it is reasonable to take C and D to be equivalent. (This example is due to Eugenio Moggi.)

Indeed, Sangiorgi [26] has defined a natural version of strong bisimilarity which achieves this equivalence, and whose induced congruence coincides with *strong reduction congruence*, $\sim_r$. Furthermore, by suitably ignoring τ actions, the corresponding *weak* bisimilarity induces a congruence which coincides with *weak reduction congruence*, $\approx_r$, as defined in Section 5.1. This alternative characterization of the reduction congruences adds to their importance.

Furthermore, it is up to weak reduction congruence that, in a precise sense, higher-order processes can be encoded as first-order processes. In Section 7.4 we give this encoding, though not the proof by Sangiorgi [26] of its faithfulness (which is not immediate). But first we must extend sorting to higher order.

7.3 Higher-order sorts

In our extension to the syntax we ignored sorts, and the reader may have felt uncomfortable – since there is so much more nonsense which can be written in the unsorted π-calculus at higher order! For example, the application $((\lambda x)\overline{x}.\mathbf{0})\mathbf{0}$ is clearly nonsense, and our sorting discipline will forbid it.

Recalling the first-order sorting discipline of Section 6.1, we only have to make a simple change. There, an object sort was a sequence of subject sorts, e.g. $(S_1 S_2 S_3)$. Now, we must allow object sorts themselves to occur in such sequences, e.g. we must allow $(S_1(S_2 S_1)S_3)$ and even $(S_1(S_2(S_1 S_2))S_3)$. In other words each element of such a sequence may be a *data sort*, where the data sorts $Dat(\mathcal{S})$ are

$$Dat(\mathcal{S}) \stackrel{\text{def}}{=} \mathcal{S} \cup Ob(\mathcal{S})$$

(a disjoint union), and we define by mutual recursion

$$Ob(\mathcal{S}) \stackrel{\text{def}}{=} Dat(\mathcal{S})^*$$

We assume that there are infinitely many abstraction variables $X, Y, \ldots$ at each object sort s, and we write $X : s$.

With these new arrangements, the formation rules of Section 6.1 remain unchanged, and we merely extend them with the following for abstractions:

$$\frac{X : s \qquad F : t}{(\lambda X)F : (s)\hat{\ }t}$$

$$\frac{F : (S)\hat{\ }t \qquad x : S}{Fx : t} \qquad\qquad \frac{F : (s)\hat{\ }t \qquad G : s}{FG : t}$$

and for concretions

$$\frac{F : s \qquad C : t}{[F]C : (s)\hat{\ }t}$$

As before, we say that A is *well-sorted* for *ob* if we can infer $A : s$ for some s from the formation rules.

Now we have introduced a rich sort discipline, comparable with the simple type-hierarchy in λ-calculus. It is important to see what is missing, and why. First, note that we might naturally have used an arrow in our sort representation, giving the following syntax for higher-order object sorts:

$$s \ ::= \ () \ \big| \ S \to s \ \big| \ s \to s$$

With this syntax it is clear that every object sort has the form

$$d_1 \to \cdots \to d_n \to ()$$

($n \geq 0$) where each d_i is a data sort. Indeed, what we have done is to choose to write this in the form

$$(d_1 \cdots d_n)$$

Thus, in our formation rules, we have written $(s)\widehat{\ }t$ for the more familiar $s \to t$.

We can now see what is missing; there are no sorts of the form

$$d_1 \to \cdots \to d_n \to S$$

Why not? What would be an inhabitant of this sort? It would take n data parameters and return a *name*. But in the presence of such calculation of names, the simple but subtle behaviour of our restriction operator (νx), which is a scoping device for names, appears irretrievably lost! For the essence of syntactic scoping is that "scoped" or "bound" occurrences of a name are syntactically manifest, and this would no longer be the case with name-calculation. This point is, of course, equally relevant to the first-order π-calculus, and it deserves further examination.

Given a sorting, and knowing the sort of each name and variable, it is easy enough to determine a unique sort – if it exists – for any agent. More intriguing is the following problem: given an agent A, but no sorting or sort information, find a sorting *ob* and an assignment of sorts to names and variables so that A is well-sorted for *ob*. This is a non-trivial problem even at first order. It remains to be seen whether there is in some sense a *most general* sorting, and how to find it.

Exercise Find a higher-order sorting *ob*, and sorts for the names x, y, z and the variable X, such that P respects *ob*, where

$$P \equiv \overline{x}.\big[(\lambda z)\overline{z}.\mathbf{0}\big] \mid x.(\lambda X)(Xy) \qquad \blacksquare$$

7.4 Translating higher order to first order

Having generalized the notion of sort, we can now see the relationship between the higher-order π-calculus and Thomsen's (Plain) CHOCS. It is rather clear. Thomsen allows processes, but not names, to be transmitted in communication; that is, every communicated datum in CHOCS has sort (). This corresponds to the sorting

$$\{\,\text{NAME} \mapsto (())\,\}$$

and we are therefore justified in regarding CHOCS as *second-order*. Notice that first-order π-calculus is not subsumed by CHOCS. But now let us define the *order* of a sorting as, simply, the maximum depth of nesting of parentheses in its object sorts. With this definition, *second order* properly includes both the (first-order)

π-calculus and CHOCS, since it admits both their sortings; it also admits other sortings, containing object sorts such as $(S(S))$.

In this section we introduce, by illustration rather than formally, a translation from processes of arbitrarily high order to first order, effectively extending that of Thomsen for Plain CHOCS. The translation, which we denote by $(\hat{\hat{\ }})$, operates both upon sorts and upon processes. There is a close operational correspondence between P and $\hat{\hat{P}}$; we shall only illustrate this correspondence rather than express it as a theorem. But first, we wish to state the theorem which expresses the faithfulness of the translation in terms of preserving congruence. The theorem was proved by Sangiorgi [26]. It holds for processes which respect a finite higher-order sorting, and which contain no free abstraction variables. We shall call these *proper* processes.

Theorem (Sangiorgi) Let ob be a finite sorting of arbitrarily high order, and let P, Q be proper processes which respect ob. Then

1. $\hat{\hat{P}}$ respects $\widehat{\widehat{ob}}$.
2. If ob is first-order then $\widehat{\widehat{ob}} = ob$ and $\hat{\hat{P}} \equiv P$.
3. $P \approx_r Q$ iff $\hat{\hat{P}} \approx_r \hat{\hat{Q}}$. ■

The translation $(\hat{\hat{\ }})$ is constructed iteratively. Each iteration applies a translation $(\hat{\ })$ both to the sorting and to the process, and the proof of the theorem proceeds by showing that the asserted results hold exactly for $(\hat{\ })$, except that $\widehat{ob}$ and $\hat{P}$ are not necessarily first-order. But if ob is finite then $\widehat{ob}$ is lower than ob, according to a clearly well-founded ordering; this completes the proof.

We illustrate $(\hat{\ })$. Suppose

$$ob \;=\; \{\, S_1 \mapsto (),\; S_2 \mapsto (S_3),\; S_3 \mapsto (S_1(S_2(S_1))((S_3)))\,\}$$

Since ob is not first-order, we first choose a highest-order data sort in a highest-order object sort in the range of ob. Let us choose $(S_2(S_1))$. Then we "depress" $(S_2(S_1))$, replacing it by a new subject sort S_4, and adding $S_4 \mapsto (S_2(S_1))$ to the sorting. Thus we obtain

$$\widehat{ob} \;=\; \{\, S_1 \mapsto (),\; S_2 \mapsto (S_3),\; S_3 \mapsto (S_1 S_4((S_3))),\; S_4 \mapsto (S_2(S_1))\,\}$$

It is clear that iterating $(\hat{\ })$ upon a finite sorting will reach first order in a finite number of steps.

Exercise What measure is decreased by $(\hat{\ })$? ■

Let us use a slightly simpler sorting

$$ob \;=\; \{\, S_1 \mapsto (),\; S_2 \mapsto ((S_1))\,\}$$

to illustrate the translation of processes. First we have

$$\widehat{ob} \;=\; \{\, S_1 \mapsto (),\; S_2 \mapsto (S_3),\; S_3 \mapsto (S_1)\,\}$$

Now let

$$P \;\equiv\; \overline{x}.[F]Q \mid x.(\lambda X)(Xy \mid Xz)$$

Then P respects ob, provided $x : S_2$, $y, z : S_1$, $X : (S_1)$, $F : (S_1)$ and $Q : ()$. We describe $\widehat{P}$ in outline. The translation only affects those subexpressions $\alpha.A$ of P whose subject is of sort S_2, since the object sort $ob(S_2) = ((S_1))$ has been changed to (S_3). In this case both x and $\overline{x}$ are involved. The appropriate changes are to replace the datum F, of sort (S_1), by a new restricted datum *name* $u : S_3$, and to abstract a new *name* $v : S_3$ in place of the abstraction variable $X : (S_1)$:

$$\widehat{P} \;\equiv\; \overline{x}.(\nu u)[u](\,!u.\widehat{F} \mid \widehat{Q}) \mid x.(\lambda v)(\overline{v}.[y] \mid \overline{v}.[z])$$

The reader may at this point like to compare the simpler example at the beginning of Section 7.1; here we are dealing with the extra complication that X occurs twice, and is moreover a variable over *abstractions*, not over *processes*. Note particularly the use of replication; since (as here) the datum F may be "used" several times by its recipient, the translation has to allow repeated access to it. This access is via u, which becomes bound to v; note how the argument to which X was applied is, in the translation, transmitted along u in a communication. One can indeed check that the reduction

$$P \;\rightarrow\; Q \mid Fy \mid Fz$$

is matched by a triple reduction in the translation:

$$\begin{array}{rcl} \widehat{P} & \rightarrow^3 & (\nu u)(\,!u.\widehat{F} \mid \widehat{Q} \mid \widehat{F}y \mid \widehat{F}z) \\ & \equiv & (\nu u)(\,!u.\widehat{F}) \mid \widehat{Q} \mid \widehat{F}y \mid \widehat{F}z \\ & \sim & \widehat{Q} \mid \widehat{F}y \mid \widehat{F}z \end{array}$$

Note that this single application of $(\widehat{\ })$ only deals with those subexpressions $\alpha.A$ in F and Q for which $\alpha : S_2$. In this example, the translation will then be first-order (because the sorting $\widehat{ob}$ is first-order). In general, the new prefixed expression $u.\widehat{F}$ may need treatment in a further iteration of $(\widehat{\ })$; noting that $u : S_3$, this would be the case if $\widehat{ob}(S_3)$ were still not first-order.

In conclusion, we can ask whether our theorem is sufficiently general. The constraint of no free abstraction variables is of no great concern. Also, we have elsewhere questioned whether there are any useful processes which respect no sorting. There remains the constraint that the sorting should be *finite*. There are indeed interesting infinite sortings; but we conjecture that, if a process respects any sorting at all, then it respects a finite sorting (and so is amenable to our translation). If this is true, then indeed the theorem is quite general.

References

[1] Abramsky, S., *The lazy lambda calculus*, To appear in **Declarative Programming**, ed. D.Turner, Addison Wesley, 1989.

[2] Astesiano, E. and Reggio, G., *SMoLCS-driven concurrent calculi*, Lecture Notes in Computer Science, Vol 249, pp169–201, Springer-Verlag, 1987.

[3] Astesiano, E. and Zucca, E., *Parametric channels via Label Expressions in CCS*, Journal of Theor. Comp. Science, Vol 33, pp45–64, 1984.

[4] Baeten, J.C.M. and Weijland, W.P., **Process Algebra**, Cambridge University Press 1990.

[5] Berry, G. and Boudol, G., *The chemical abstract machine*, Proc 17th Annual Symposium on Principles of Programming Languages, 1990.

[6] Boudol, G., *Towards a lambda-calculus for concurrent and communicating systems*, Proc. TAPSOFT 89, Lecture Notes in Computer Science, Vol 351, pp149–161, Springer-Verlag, 1989.

[7] Burstall, R.M., *Some techniques for proving correctness of programs which alter data structures*, in **Machine Intelligence 7**, ed. B.Meltzer and D.Michie, Edinburgh University Press, pp23–50, 1972.

[8] Engberg, U. and Nielsen, M., *A Calculus of Communicating Systems with Label-passing*, Report DAIMI PB–208, Computer Science Department, University of Aarhus, 1986.

[9] Girard, J.-Y., *Linear logic*, J. Theoretical Computer Science, Vol 50, pp1–102, 1987.

[10] Hennessy, M., **Algebraic Theory of Processes**, MIT Press, 1988.

[11] Hennessy, M. and Milner, R., *Algebraic laws for non-determinism and concurrency*, Journal of ACM, Vol 32, pp137–161, 1985.

[12] Hewitt, C., Bishop, P. and Steiger, R., "A Universal Modular Actor Formalism for Artificial Intelligence", Proc IJCAI '73, Stanford, California, pp235–245, 1973.

[13] Hoare, C.A.R., **Communicating Sequential Processes**, Prentice Hall, 1985.

[14] Kennaway, J.R. and Sleep, M.R., *Syntax and informal semantics of DyNe, a parallel language*, Lecture Notes in Computer Science, Vol 207, pp222–230, Springer-Verlag, 1985.

[15] Milner, R., **A Calculus of Communicating Systems**, Lecture Notes in Computer Science, Volume 92, Springer-Verlag, 1980.

[16] Milner, R., **Communication and Concurrency**, Prentice Hall, 1989

[17] Milner, R., *Functions as processes*, Research Report No. 1154, INRIA, Sophia Antipolis, February 1990. To appear in Journal of Mathematical Structures in Computer Science.

[18] Milner, R., Sorts in the π-calculus, Proc. Third Workshop on Concurrency and Compositionality, Goslar, Germany; to appear as a volume of Springer Verlag Lecture Notes in Computer Science, 1991.

[19] Milner, R., Parrow, J. and Walker D., *A calculus of mobile processes*, Reports ECS-LFCS–89–85 and –86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989. (To appear in Journal of Information and Computation.)

[20] Milner, R., Parrow, J. and Walker, D., *Modal logics for mobile processes*, Report ECS-LFCS–91–136, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1991. (To appear in Proceedings of CONCUR '91, Amsterdam.)

[21] Milner, R. and Sangiorgi, D., *Barbed Bisimulation*, Internal memorandum, Computer Science Dept., University of Edinburgh, 1991.

[22] Nielson, F., *The typed λ-calculus with first-class processes*, Proc. PARLE 89, Lecture Notes in Computer Science, Vol 366, Springer-Verlag, 1989.

[23] Orava, F. and Parrow, J., *An algebraic verification of a mobile network*, Internal report, SICS, Sweden, 1990. To appear in Journal of Formal Aspects of Computer Science.

[24] Park, D.M.R., *Concurrency and automata on infinite sequences*, Lecture Notes in Computer Science, Vol 104, Springer Verlag, 1980

[25] Petri, C.A., Fundamentals of a theory of asynchronous information flow, Proc. IFIP Congress '62, North Holland, pp386–390, 1962.

[26] Sangiorgi, D., Forthcoming PhD thesis, University of Edinburgh, 1992.

[27] Thomsen, B., *Calculi for higher-order communicating systems*, PhD thesis, Imperial College, London University, 1990.

[28] Walker, D.J., *π-calculus semantics of object-oriented programming Languages*, Report ECS-LFCS–90–122, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1990. Proc. Conference on Theoretical Aspects of Computer Software, Tohoku University, Japan in September 1991.