A Calculus of Mobile Processes, Part I

Robin Milner, University of Edinburgh, Scotland
Joachim Parrow, Swedish Institute of Computer Science, Kista, Sweden
David Walker, University of Technology, Sydney, Australia

June 1989 (Revised September 1990)

Running title: Calculus of Mobile Processes, Part I

Address for proofs: Robin Milner, Computer Science Department, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, UK

Abstract: We present the π-calculus, a calculus of communicating systems in which one can naturally express processes which have changing structure. Not only may the component agents of a system be arbitrarily linked, but a communication between neighbours may carry information which changes that linkage. The calculus is an extension of the process algebra CCS, following work by Engberg and Nielsen who added mobility to CCS while preserving its algebraic properties. The π-calculus gains simplicity by removing all distinction between variables and constants; communication links are identified by *names*, and computation is represented purely as the communication of names across links.

After an illustrated description of how the π-calculus generalises conventional process algebras in treating mobility, several examples exploiting mobility are given in some detail. The important examples are the encoding into the π-calculus of higher-order functions (the λ-calculus and combinatory algebra), the transmission of processes as values, and the representation of data structures as processes.

The paper continues by presenting the algebraic theory of *strong bisimilarity* and *strong equivalence*, including a new notion of equivalence indexed by *distinctions* – i.e. assumptions of inequality among names. These theories are based upon a semantics in terms of a labelled transition system and a notion of *strong bisimulation*, both of which are expounded in detail in a companion paper. We also report briefly on work-in-progress based upon the corresponding notion of *weak* bisimulation, in which internal actions cannot be observed.

Special symbols: Since this copy is mathematically type-set, only a few of the less obvious symbols are listed below.

$\alpha\ \beta\ \lambda\ \pi\ \sigma\ \tau$	Greek letters
$= \equiv \sim \quad \dot{\sim} \quad \approx \quad \dot{\approx} \quad \simeq \quad \dot{\simeq} \quad \lesssim \quad \gtrsim \quad \overline{\overline{\sim}}$	Various relation symbols
$\mathbf{0}$	Boldface zero
$\emptyset$	Empty set
$\mathbf{S}\ \mathbf{S}_0\ \mathbf{S}_1\ \mathbf{S}_2\ \mathbf{K}\ \mathbf{K}_0\ \mathbf{K}_1\ \mathbf{I}\ \mathbf{I}_0\ \mathbf{I}_1$	Boldface symbols for Combinators
$\textsc{s}_0\ \ldots\ \textsc{t}\ \ldots\ \textsc{cons}$	Various constant names
A C0 ... **SGE**	Boldface names of axioms etc
$[\![\ \ ]\!]$	Double square brackets

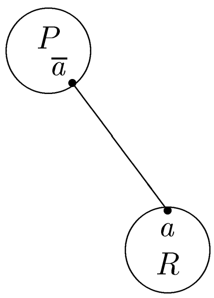
1 Introduction

We present a calculus of communicating systems in which one can naturally express processes which have changing structure. Not only may the component agents of a system be arbitrarily linked, but a communication between neighbours may carry information which changes that linkage.

The most mathematically developed models of concurrency can at best express this *mobility* – as we shall call it – indirectly. Examples are: Petri nets [14], CSP [7], ACP [3], CCS [9]. On the other hand there are models which express mobility directly but which still require, in our view, a mathematical analysis of their basic concepts such as we provide in this paper. A well-known model of this kind, which has had considerable success in applications, is the Actors model of Hewitt [5]. In such models, mobility is often achieved by allowing processes to be passed as values in communication; we shall instead achieve it by allowing *references* to processes, i.e. links, to be communicated. This presents an interesting contrast with recent attempts to combine the ideas of λ-calculus and process calculi by admitting processes as values; examples are by Gerard Boudol [4], Flemming Nielson [12] and Bent Thomsen [16].

The calculus given here is based upon the approach of Uffe Engberg and Mogens Nielsen [6], who successfully extended CCS to include mobility while preserving its algebraic properties. In the concluding section we describe in more detail what we have added to that work; roughly speaking, we retain (we hope) its essence, but reduce its complexity and strengthen its elementary theory.

We shall introduce the calculus by means of a sequence of examples, which are clearly of practical significance and which fall naturally into the formalism. Let us begin with a very simple example; we present it at first in the notation of CCS, and we shall use informally a kind of diagram, which we call a *flow graph*, to represent the linkage between (or among) agents. We suppose that an agent P wishes to send the value 5 to an agent R, along a link named a, and that R is willing to receive any value along that link. Then the appropriate flow graph is as follows:

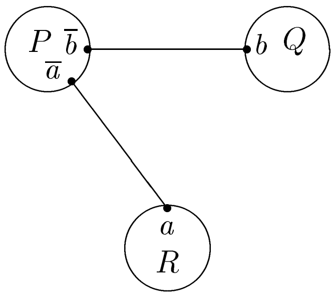


We may have, for example, $P \equiv \overline{a}5.P'$ and $R \equiv a(x).R'$. The prefix $a(x)$ binds the variable x in R'; in general, both here and later, we use parentheses to indicate the binding occurrence of a variable. The system depicted in the flow graph is represented by the expression

$$(\overline{a}5.P' \mid a(x).R')\backslash a$$

The postfixed operator $\backslash a$ is called a *restriction*, and indicates that the link a is private to P and R.

Let us now suppose instead that P wishes to delegate to a new agent, Q, the task of transmitting 5 to R. We therefore suppose that P is connected to Q initially by a link b:



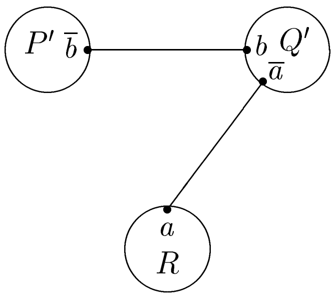
We now let $P \equiv \overline{b}a.\overline{b}5.P'$; it sends along b both the link a and the value 5 to be transmitted along a. We also let $Q \equiv b(y).b(z).\overline{y}z.\mathbf{0}$; it receives a link and a value upon b, then transmits the value along the link and terminates. Note that the name a is not in the expression Q; Q possesses no link to R initially. The whole system is now

$$(\overline{b}a.\overline{b}5.P' \mid b(y).b(z).\overline{y}z.\mathbf{0} \mid a(x).R')\backslash a\backslash b$$

After two communications, both along b, it then becomes

$$(P' \mid \overline{a}5.\mathbf{0} \mid a(x).R')\backslash a\backslash b$$

Thus, if a does not appear in P', we may draw the new configuration of the system as follows, indicating that P's a-link has moved to Q, and Q has become $Q' \equiv \overline{a}5.\mathbf{0}$:



This formalism, in which link names appear as parameters in communication, goes beyond CCS. It may seem that with the addition of variables over link names, as well as over ordinary data values, the calculus would become over-rich in primitives. But we shall avoid this prodigality. In fact we shall remove all distinction among link names, variables and ordinary data values; we shall call them all *names*. There will be just two essential classes of entity: names and agents. Restriction and input-prefix become name-binding operators of different nature; restriction localises the scope of a name, while input-prefix is similar to abstraction in the λ-calculus (being a place-holder for a name which may be received as input). To emphasize the name-binding property of restriction we shall write $(x)P$ in place of $P\backslash x$; with this syntax, the above example becomes

$$(a)(b)(\overline{b}a.\overline{b}5.P' \mid b(y).b(z).\overline{y}z.\mathbf{0} \mid a(x).R')$$

Note that $a, b, x, y, 5$ are *all* just names.

It will appear as though we reduce all concurrent computation to something like a cocktail party, in which the only purpose of communication is to transmit (or to receive) a name which will admit further communications. Surprisingly, this meagre basis is enough to encode computation over an arbitrary data types, if we consider a data type to be a set of *data structures* – values recursively built from a given finite set of constructors. We tentatively call our new calculus *the π-calculus*, since it aims for universality (at an elementary level) for concurrent computation, just as the λ-calculus is universal for functional computation.

In a companion paper [11] we treat the semantics of the π-calculus in depth. The present paper is devoted to a sequence of motivating examples, followed by a statement of the important algebraic properties. In more detail, the remainder of the paper proceeds as follows. In Section 2 we define the constructions of the π-calculus with some auxiliary notions; we then discuss its salient differences from CCS. In Section 3 we look at some basic examples of the calculus; these are simple finite processes which indicate how scope and mobility are closely interdependent notions. In Section 4, we introduce some convenient abbreviations, which allow us to treat more realistic examples. In particular, we carefully compare the passing of names as parameters with the passing of processes as parameters; we also show how to encode data structures as processes. This section should indicate, particularly to those familiar with process algebras, that the addition of names-as-parameters to CCS provides great modelling strength and transforms the nature of these algebras.

Some of the examples in Section 4 are quite substantial; the reader may safely skip some or all of them on a first reading, and proceed to Section 5 without loss of continuity.

In Section 5 we present the equational theory of bisimilarity, as it is defined and derived in the companion paper [11]. Although this equational theory is strikingly simple, one feature is noteworthy, and needs careful treatment; it is that bisimilarity is not preserved in general by instantiation of names. Our solution appears to be quite tractable; it is to adopt a relativized equality, which is preserved only by those instantiations which maintain the distinction between certain pairs of names. We derive some convenient laws for this relativized equality. The section also records the fact that the equational theory of *weak equality*, in which the internal τ actions of a system are ignored as far as possible, is a direct generalization from that in CCS.

2 The calculus

We presuppose an infinite set $\mathcal{N}$ of *names*, and let u, v, w, x, y, z range over names. We also presuppose a set $\mathcal{K}$ of *agent identifiers*, each with an *arity* – an integer ≥ 0. We let $A, B, C, \ldots$ range over agent identifiers. We now let $P, Q, R, \ldots$ range over the *agents* or *process expressions*, which are of six kinds as follows:

1. A *summation* $\sum_{i \in I} P_i$, where the index set I is finite.

   This agent behaves like one or other of the P_i. We write $\mathbf{0}$ for the empty summation, and call it *inaction*; this is the agent which can do nothing. Henceforward, in defining the calculus, we confine ourselves just to $\mathbf{0}$ and binary summation, written $P_1 + P_2$.

2. A *prefix* form $\overline{y}x.P$, $y(x).P$ or $\tau.P$.

   '$\overline{y}x.$' is called a *negative* prefix. $\overline{y}$ may be thought of as an *output port* of an agent which contains it; $\overline{y}x.P$ outputs the name x at port $\overline{y}$ and then behaves like P.

   '$y(x).$' is called a *positive* prefix. A name y may be thought of as an *input port* of an agent; $y(x).P$ inputs an arbitrary name z at port y and then behaves like $P\{z/x\}$ (see the definition of substitution below). The name x is bound by the positive prefix '$y(x).$'. (Note that a negative prefix does not bind a name.)

   '$\tau.$' is called a *silent* prefix. $\tau.P$ performs the *silent action* τ and then behaves like P.

3. A *composition* $P_1 \mid P_2$.

   This agent consists of P_1 and P_2 acting in parallel. The components may act independently; also, an output action of P_1 (resp. P_2) at any output port $\overline{x}$ may synchronize with an input action of P_2 (resp. P_1) at x, to create a silent (τ) action of the composite agent $P_1 \mid P_2$.

4. A *restriction* $(x)P$.

   This agent behaves like P except that actions at ports $\overline{x}$ and x are prohibited (but communication *between* components of P along the link x are not prohibited, since they have become τ actions as explained above).

5. A *match* $[x = y]P$.

   This agent behaves like P if the names x and y are identical, and otherwise like $\mathbf{0}$.

6. A *defined agent* $A(y_1, \ldots, y_n)$.

For any agent identifier A (with arity n) used thus, there must be a unique *defining equation* $A(x_1, \ldots, x_n) \stackrel{\mathrm{def}}{=} P$, where the names $x_1, \ldots, x_n$ are distinct and are the only names which may occur free in P. Then $A(y_1, \ldots, y_n)$ behaves like $P\{y_1/x_1, \ldots, y_n/x_n\}$ (see below for the definition of substitution). Defining equations provide recursion, since P may contain any agent identifier, even A itself.

The syntax of agents may be summarized as follows:

$$
\begin{array}{rcl}
P & ::= & \mathbf{0} \\
& | & P_1 + P_2 \\
& | & \overline{y}x.P \\
& | & y(x).P \\
& | & \tau.P \\
& | & P_1 \mid P_2 \\
& | & (x)P \\
& | & [x = y]P \\
& | & A(y_1, \ldots, y_n)
\end{array}
$$

When our attention is confined to *finite* agents, i.e. agents with finite behaviour, the agent identifiers and their definitions can be omitted, thus removing recursion. The π-calculus without the match form is also interesting. Although matching makes it easy to encode computation with data structures, it turns out to be unnecessary for this purpose, as we shall see in Section 4, Example 7. We include the match form partly for clarity, and partly for the pleasant form of expansion law which it provides (rule $\mathbf{E}'$ in Section 5).

A few further definitions will be needed.

- The *free names* $\mathrm{fn}(P)$ of P are just those names which occur in P not bound either by a positive prefix or by a restriction. We write $\mathrm{fn}(P_1, P_2, \ldots)$ for $\mathrm{fn}(P_1) \cup \mathrm{fn}(P_2) \cup \cdots$.

- As in the λ-calculus we shall not distinguish between agents which are *alpha-convertible*, i.e. which differ only by a change of bound names. We shall write $P \equiv Q$ if P and Q are alpha-convertible.

- We shall sometimes write $\tilde{x}$ for the vector $\{x_1, \ldots, x_n\}$ of names, where n is understood from context.

- We write $P\{y_1/x_1, \ldots, y_n/x_n\}$, or $P\{y_i/x_i\}_{1 \le i \le n}$, or $P\{\tilde{y}/\tilde{x}\}$, for the simultaneous substitution of y_i for all free occurrences of x_i (for $1 \le i \le n$) in P, with change of bound names if necessary to prevent any of the new names y_i from becoming bound in P.

- In the prefixes '$\overline{y}x.$' and '$y(x).$' we say that y is the *subject*, and x is the *object* or *parameter*. It is the names occurring free in subject position which determine the (current) communication capabilities of an agent.

- We adopt the following precedence among the syntactic forms:

$$\left.\begin{array}{r}\text{Restriction}\\ \text{Prefix}\\ \text{Match}\end{array}\right\} > \text{Composition} > \text{Summation}$$

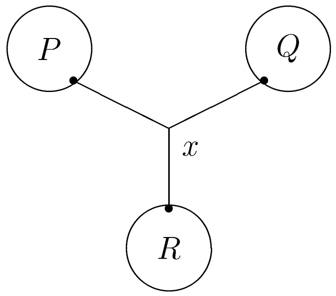
  Thus, $(x)P \mid \tau.Q + R$ means $(((x)P) \mid (\tau.Q)) + R$.

We now discuss some of the more important features of the calculus, as a preparation for the examples in the following two sections.

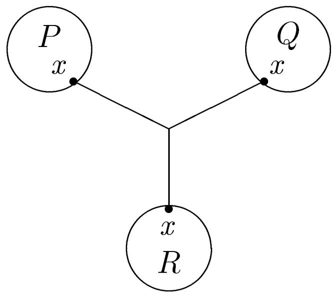
1. Apart from the presence of parallel composition, the outstanding difference between the π-calculus and the λ-calculus is that names may be instantiated only to names, not to agents (i.e. expressions). This distinction will be understood better through our examples. We explain informally here why we have chosen this course. An agent enacts a process, which changes state through its history; parallel composition admits communication among a family of such processes each with independent state. Such a communication is typically formed by the synchronization of a negative prefix '$\overline{y}x.$' with a positive prefix '$y(z).$'. As an example, suppose the sending agent is $P \equiv \overline{y}x.P'$, and the receiving agent is $Q \equiv y(z).Q'$; then Q may acquire from P access – via x – to an already existing agent R (see our first example below). One may choose to model this by passing R *itself* as a parameter, rather than just passing an x-link as a parameter, but we choose not to do so for several reasons.

   First, to pass R as a parameter to Q may result in replication of R, due to repeated occurrence of the formal parameter z within Q'; we do not wish the replication of agents with state to be a side-effect of communication in the π-calculus. Second, to pass R as a parameter gives Q access to the *whole* of R; we are concerned to model the case where a received name x provides only *partial* access to another agent. (For example, R may communicate with still other neighbours via names not known to Q.) Third, the transmission of access links is a very common phenomenon in computation, even in purely sequential computation, which has hitherto had no adequate theoretical basis; we must examine primitives which may provide such a basis in as lean a framework as possible.

2. The free names of an agent represent its current knowledge of, or linkage to, other agents. If several agents – say P, Q and R – all contain x free, we shall portray this linkage or shared knowledge by a *multi-arc* in the flow graph of $P \mid Q \mid R$, as follows:



Further, if the shared name x is restricted we shall write it internally to the components; so the flow graph of $(x)(P \mid Q \mid R)$ is



The free names $\mathrm{fn}(P)$ correspond roughly to what is called in CCS the *sort* of P. But there are differences. First, a sort in CCS contains both names and *co-names* such as $\overline{b}$; if $\{a, \overline{b}\}$ is the sort of a CCS agent P, it means that P may input on the link a and output on the link b, but not vice versa. For the present calculus, we do not yet wish to adopt this refinement. Second, in CCS a sort cannot grow throughout the history of an agent; however, the free names $\mathrm{fn}(P)$ *can* grow throughout P's subsequent history, since P can receive names in communication. Third, a free occurrence of x in subject position in P indicates that P may communicate along the link x, while a free occurrence in object position merely indicates that P may pass x as a parameter; it is only the former type of occurrence which corresponds closely to CCS sort, since CCS did not admit the latter type.

This subject–object distinction is not captured by $\mathrm{fn}(P)$. In later development the distinction will no doubt become increasingly important, in order to understand the potential mobility among the agents of a system.

3. In CCS there is no risk of confusing the binding of an ordinary data variable x, in a positive prefix form $a(x).P$, with the restriction of a port-name a as in $Q\backslash a$. This is because port-names are distinct from data variables in CCS. Here, the two are identified. (More accurately, we shall see that primitive values are represented as names, while compound values – e.g. trees – are represented as processes.) To emphasize this identification, we have adopted the prefix notation $(x)Q$ for restriction, in place of the postfix

notation $Q\backslash x$. Nonetheless, we must carefully distinguish between the two forms of binding, $y(x).P$ and $(x)Q$. In $y(x).P$, x is a place-holder for any name which may be received on the link y, and this may even be a name already free in P; thus the variable bound by a positive prefix is susceptible to arbitrary instantiation. In $(x)Q$, x represents a name which is private to Q, and which moreover can never be identified with any other name in Q. The different treatment of these bindings lies at the heart of the π-calculus.

In this paper we do not present the basic semantics of the calculus; this is done in our companion paper [11], in the same style as in CCS, namely as a labelled transition system defined by structural inference rules. In that paper the notions of strong bisimulation and strong equivalence are also defined; the latter is a congruence relation, so it may be understood as (strong) semantic equality. Here, we shall rely somewhat upon analogy with the transitions of CCS agents. In particular, we shall assume simple transitions such as

$$(\cdots + \overline{y}x.P + \cdots) \mid (\cdots + y(z).Q + \cdots) \quad \xrightarrow{\tau} \quad P \mid Q\{x/z\}$$

and simple equations such as

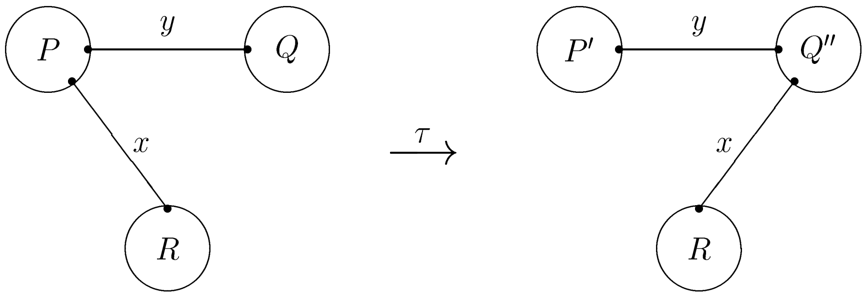
$$(y)(\overline{y}x.P \mid y(z).Q) \quad = \quad \tau.(y)(P \mid Q\{x/z\})$$

in which $=$ means strong equivalence. But not all transitions will be analogous to CCS; the reader will find the essence of mobility in Example 3 in the next section, where we see that the effect of certain transitions is to change the scope of a restriction.

3 Basic examples

The examples in this section are almost entirely concerned with different patterns of occurrence of the two forms of name-binding, positive prefix and restriction, and their behaviour in the presence of communication. This is the basic anatomy of the π-calculus.

Example 1: Link passing

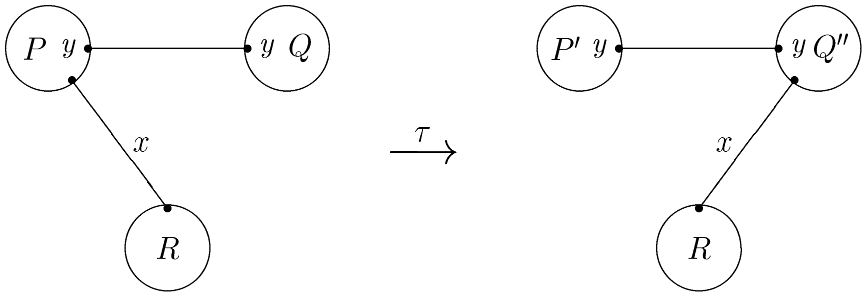


The agent P has a link x to R, and wishes to pass x along its link y to Q. Q is willing to receive it. Thus P may be $\overline{y}x.P'$ and Q may be $y(z).Q'$; in this case, the transition is

$$\overline{y}x.P' \mid y(z).Q' \mid R \xrightarrow{\tau} P' \mid Q'\{x/z\} \mid R \tag{1}$$

So Q'' in the diagram is $Q'\{x/z\}$. The diagram illustrates the case in which $x \notin \mathrm{fn}(Q)$, meaning that Q possesses no x link before the transition. But the transition is the same if $x \in \mathrm{fn}(Q)$; there is no reason that Q should not receive a link which it already possesses. (Compare Examples 2 and 3, though, in which one or other of the x links is private.) The diagram also illustrates the case in which $x \notin \mathrm{fn}(P')$, meaning that P' has no x-link after the transition, but again this condition does not affect the transition.

The situation is not much different when the link y between P and Q is private. In this case the proper flow graphs are



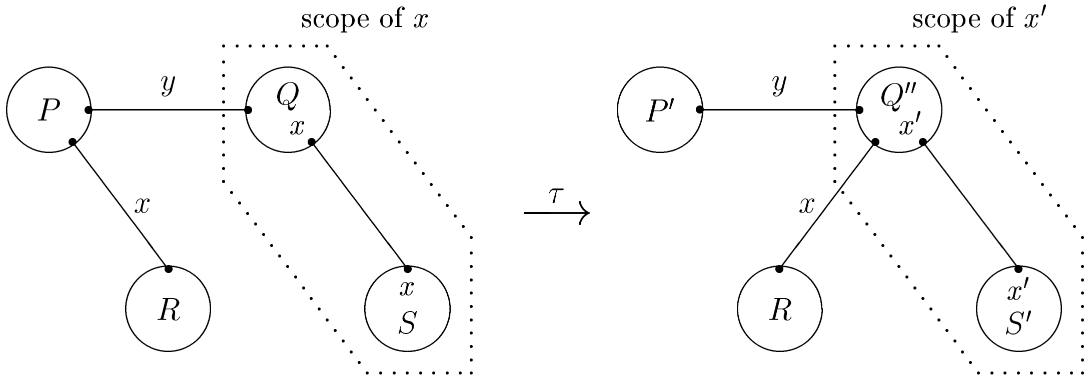
The privacy of the y-link is represented by a restriction, so the transition is now

$$(y)(\overline{y}x.P' \mid y(z).Q') \mid R \xrightarrow{\tau} (y)(P' \mid Q'\{x/z\}) \mid R \tag{2}$$

In fact we shall be able to prove the equation

$$(y)(\overline{y}x.P' \mid y(z).Q') = \tau.(y)(P' \mid Q'\{x/z\}) \tag{3}$$

Example 2: Scope intrusion



As in Example 1, P has a link x to R, and wishes to pass x along its link y to Q. Q is willing to receive it, but already possesses a private link x to S; the latter must be renamed to avoid confusion. We describe this informally by saying that P (or the link-passing action) *intrudes the scope* of the private link x between Q and S.

Taking P to be $\overline{y}x.P'$ and Q to be $y(z).Q'$ as in Example 1, the transition is

$$\overline{y}x.P' \mid R \mid (x)(y(z).Q' \mid S) \xrightarrow{\tau} P' \mid R \mid (x')(Q'\{x'/x\}\{x/z\} \mid S\{x'/x\}) \tag{4}$$

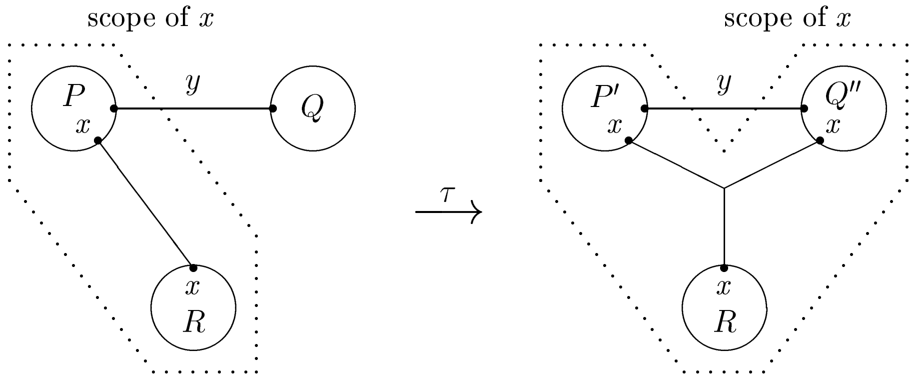
So Q'' and S' in the diagram are $Q'\{x'/x\}\{x/z\}$ and $S\{x'/x\}$ respectively, where x' is a new name.

This phenomenon is analogous to the avoidance of the capture of bound variables in the λ-calculus. The transition rules will be such that, as in the λ-calculus, *alpha-conversion* (i.e. change of bound variables) is enforced in such cases. For the present example, the transition rules will ensure that the transition (4) is the same, up to alpha-conversion of the result, as we would obtain if we applied the alpha-equivalence

$$(x)(Q \mid S) = (x')(Q\{x'/x\} \mid S\{x'/x\}) \tag{5}$$

beforehand, thus avoiding the intrusion. (Also note that, as we state in Section 5 below, alpha-equivalence implies strong equivalence of agents.)

Example 3: Scope extrusion



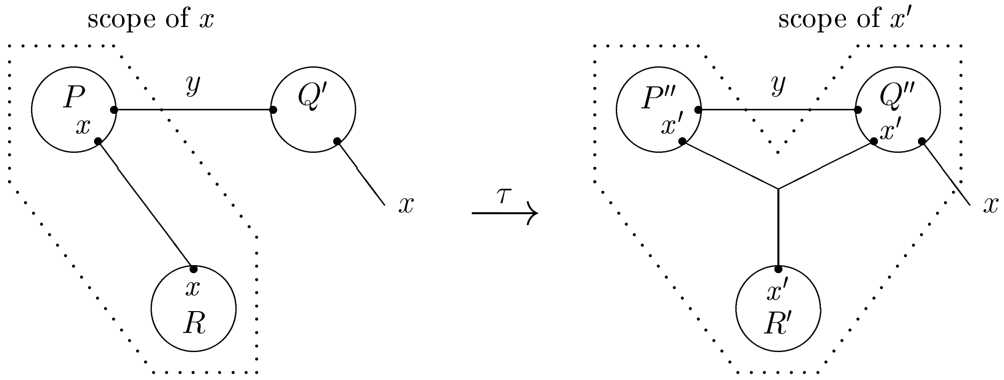
As in Example 1, P has a link x to R, but we now suppose that this link is private. However, P wishes to pass x along its link y to Q. Q is willing to receive it, and possesses no x-link. This situation is exactly that of a program P, with a *local* variable modelled by a storage register R, passing R to a procedure Q which takes its parameter *by reference*, not *by value*.

So, taking P to be $\overline{y}x.P'$ and Q to be $y(z).Q'$, as in Example 1, the transition is

$$(x)(\overline{y}x.P' \mid R) \mid y(z).Q' \xrightarrow{\tau} (x)(P' \mid R \mid Q'\{x/z\}) \tag{6}$$

So Q'' in the diagram is $Q'\{x/z\}$, as it was in Example 1. The difference here is in the privacy of P's x-link to R, represented by the restriction (x). When this link is exported to Q the scope of the restriction is extended; we say that P (or the link-passing action) *extrudes the scope* of the private x-link.

Now, in contrast with Example 1, the situation is different if Q possesses a public x-link before the transition, i.e. if $x \in \mathrm{fn}(Q)$. Then we have to change the private link name in order to preserve its distinction from the public link.

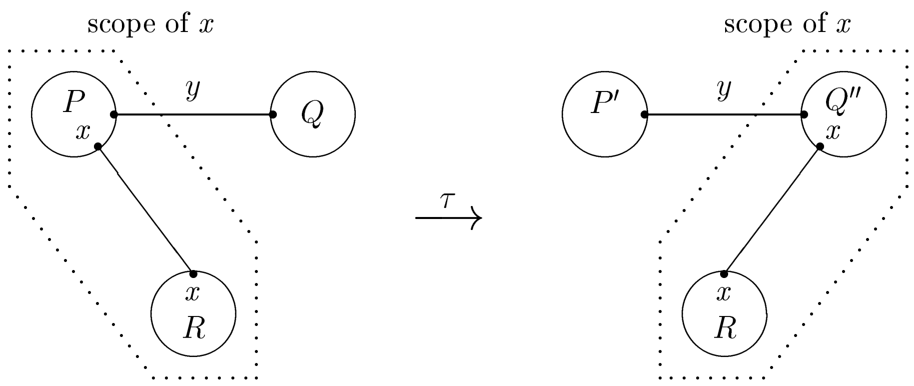


Now the transition becomes

$$(x)(\overline{y}x.P' \mid R) \mid y(z).Q' \xrightarrow{\tau} (x')(P'\{x'/x\} \mid R\{x'/x\} \mid Q'\{x'/z\}) \tag{7}$$

So P'' in the diagram is $P'\{x'/x\}$, Q'' is $Q'\{x'/z\}$, and R' is $R\{x'/x\}$.

Reverting to the case $x \notin \mathrm{fn}(Q)$, let us now suppose also that $x \notin \mathrm{fn}(P')$, i.e. P' possesses no private x-link after the transition:



The transition is exactly as in (6), but we can transform the result. There is a general law

$$(x)(P_1 \mid P_2) = P_1 \mid (x)P_2 \qquad \text{if } x \notin \mathrm{fn}(P_1) \tag{8}$$

which we can apply to the result of (6):

$$(x)(P' \mid R \mid Q'\{x/z\}) = P' \mid (x)(R \mid Q'\{x/z\}) \tag{9}$$

This is what justifies the preceding diagram. We may describe this as a *migration of scope*, a special case of extrusion; the scope of the restriction (x) has migrated from P to Q.

The reader may like to consider a combination of Examples 2 and 3, in which both intrusion and extrusion occur.

Example 4: Molecular actions Suppose that an agent P wishes to pass a pair (u, v) of names to one of two agents Q and R. Consider the following attempt at defining the three agents:

$$\begin{aligned} P &\equiv \overline{x}u.\overline{x}v.P' \\ Q &\equiv x(y).x(z).Q' \\ R &\equiv x(y).x(z).R' \end{aligned}$$

A difficulty arises, with these definitions, in the behaviour of the composite agent $Q|P|R$. It may perform the following pair of transitions, which represents the possibility that Q receives u and R receives v, instead of one or other of them receiving the *whole* pair as intended:

$$\begin{aligned} Q|P|R &\equiv x(y).x(z).Q' \mid \overline{x}u.\overline{x}v.P' \mid x(y).x(z).R' \\ &\xrightarrow{\tau} x(z).Q'\{u/y\} \mid \overline{x}v.P' \mid x(y).x(z).R' \\ &\xrightarrow{\tau} x(z).Q'\{u/y\} \mid P' \mid x(z).R'\{v/y\} \end{aligned}$$

If this possibility is to be avoided, we need a way in which the pair (u, v) of names can somehow be transmitted in a *single* atomic action. Private names are the key to the solution. Instead of passing the elements u and v directly, we arrange that P passes to Q (or to R) the private name of a small process whose only task is to deliver u and then v:

$$\begin{array}{rcl} P & \equiv & (w)(\overline{x}w.P' \mid \overline{w}u.\overline{w}v.\mathbf{0}) \\ Q & \equiv & x(w).w(y).w(z).Q' \\ R & \equiv & x(w).w(y).w(z).R' \end{array}$$

where $w \notin \mathrm{fn}(P', Q', R')$. Now, $Q|P|R$ has two alternative transitions:

$$\begin{array}{rcl} Q|P|R & \stackrel{\tau}{\to} & (w)(w(y).w(z).Q' \mid P' \mid \overline{w}u.\overline{w}v.\mathbf{0}) \mid R \\ & = & \tau.\tau.(Q'\{u/y\}\{v/z\} \mid P' \mid R) \end{array}$$

and

$$\begin{array}{rcl} Q|P|R & \stackrel{\tau}{\to} & Q \mid (w)(P' \mid \overline{w}u.\overline{w}v.\mathbf{0} \mid w(y).w(z).R') \\ & = & \tau.\tau.(Q \mid P' \mid R'\{u/y\}\{v/z\}) \end{array}$$

(We have slightly simplified these transitions, taking advantage of the associativity of $|$.) The two transitions represent the transmission of the pair to Q and to R respectively; no mixture is possible.

We may think of the atomic action $\overline{x}w$, together with the actions of the process $\overline{w}u.\overline{w}v.\mathbf{0}$ which it makes accessible, as together forming a *molecular* action. (It is vital to this idea that w, which *bonds* the molecule, is indeed a private name.) This device is very powerful, extending far beyond this illustration with pairs. As we shall see in Example 7, it yields a uniform encoding of data structures as processes.

4 Further examples

In this section we shall explore some more concrete examples; they are on a small scale, but deal with real applications in computing. First, we define some abbreviations.

1. Sometimes a communication needs to carry no parameter. To model this we presuppose a special name, say ϵ, which is never bound; then we write

$$\begin{array}{lll} \overline{x}.P & \text{in place of} & \overline{x}\epsilon.P \\ x.P & \text{in place of} & x(y).P \qquad (y \text{ not free in } P) \end{array}$$

2. We shall often omit '$.\mathbf{0}$' in an agent, and write for example

$$\overline{x}y \;\text{ in place of }\; \overline{x}y.\mathbf{0}$$

3. We shall often wish to allow input names to determine the course of computation. Thus, we naturally write

$$x(v).([v\!=\!y_1]P_1 + [v\!=\!y_2]P_2 + \cdots)$$

where usually the names y_i will be distinct. Assuming that v does not occur free in any P_i, we shall abbreviate this to

$$x : [y_1 \Rightarrow P_1,\ y_2 \Rightarrow P_2,\ \ldots]$$

or – schematically –

$$x : [y_i \Rightarrow P_i]_{i \in I}$$

Example 5: An Executor Let us define

$$Exec(x) \stackrel{\text{def}}{=} x(y).\overline{y} \tag{10}$$

$Exec(x)$ may be called an *executor*. It receives, on link x, a link which it calls y; it then activates that link. We can think of y as the *trigger* of a process which $Exec(x)$ has been called upon to run.

Now for any process P, we should (up to a few initial communications) obtain the same behaviour in each of the following cases: (a) We run P directly; (b) We prefix a trigger z to P, and pass z along the link x to the executor $Exec(x)$. (We assume $x, z \notin \text{fn}(P)$.) Here is the agent which, in the presence of $Exec(x)$, should behave like P:

$$(z)(\overline{x}z \mid z.P) \tag{11}$$

(Later we shall find that a construction like this can be regarded as passing the process P *itself* as a value along the link x, but that the passing of links as values has other applications too.) Here then is the agent which should be equivalent to P:

$$(x)((z)(\overline{x}z \mid z.P) \mid Exec(x)) \tag{12}$$

To see this, first apply equation (8) to obtain

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

Now this, by a suitably generalized expansion law, becomes

$$\tau.(x)(z)(\mathbf{0} \mid z.P \mid \overline{z})$$

which in turn becomes

$$\tau.\tau.(x)(z)(\mathbf{0} \mid P \mid \mathbf{0})$$

which, since x and z were chosen not free in P, is equal to

$$\tau.\tau.P$$

Example 6: Passing processes as parameters In the previous example, the executor had no work to do except to activate (the link to) P, and the sender had no work to do except to transmit (the link to) P (and then to retain P awaiting activation). If the parenthetical parts of the preceding sentence are included, the sentence accurately describes Example 5; if they are omitted, then it describes the passing of a process as a parameter. Though these two situations are not – at least not obviously – the same, the effect of process-passing can in many cases be achieved by link-passing.

Passing processes as parameters is not represented directly in the π-calculus. In a direct representation we would write, instead of (12), something like

$$(x)(\overline{x}P.\mathbf{0} \mid x(p).p) \tag{13}$$

where p is a variable over processes, and P is a process expression (i.e. an agent). This notation is close to that of Thomsen [16], for example (see our later discussion of his work). We have seen that, in this simple case, (12) indeed has the effect which would be intended for (13).

Let us pursue this direct representation of process-passing further, to draw attention to an important issue of scope. To develop (13) a little, suppose that the sender, after sending P, wishes to run Q; suppose also that the executor, after receiving P, wishes to run it in parallel with R. In an extended language, permitting (13), we would write

$$(x)(\overline{x}P.Q \mid x(p).(p \mid R)) \tag{14}$$

where we assume that $x \notin \mathrm{fn}(P, Q, R)$. A suitably generalized expansion law would equate this to

$$\tau.(Q \mid (P \mid R)) \tag{15}$$

This allows communication to occur both between P and it first 'neighbour', Q, and also between P and its second 'neighbour', R.

To develop the example further, we now suppose that before transmission a private link w exists between P and Q; this privacy may be represented as a restriction (w) applied to the sender:

$$(x)((w)(\overline{x}P.Q) \mid x(p).(p \mid R)) \tag{16}$$

Now there are two alternatives for how the transmission of P should treat the private link w; the choice is significant even when $w \notin \mathrm{fn}(R)$, and even more significant when $w \in \mathrm{fn}(R)$.

In the first alternative, the generalized expansion law would equate (16) with

$$\tau.((w)Q \mid (P \mid R)) \tag{17}$$

This shows that the private link w between P and Q is broken by the communication. To put it differently, in the expression $(w)(\overline{x}P.Q)$, the restriction (w) binds w in Q but not in P (and thus the private link does not in fact exist!). Moreover, if $w \in \mathrm{fn}(R)$, then w represents a link between P and R. In this approach, the passing of processes as parameters amounts to passing the *text* of the process as a parameter, which is similar to the treatment of function parameters in LISP as originally defined by McCarthy. This has often been called 'dynamic binding'; the free variables in (the text of) a function parameter are interpreted in the receiving environment. Thomsen [16] has adopted dynamic binding in his Calculus of Higher-order Communicating Systems (CHOCS), and has found that many important computational phenomena can thereby be modelled satisfactorily. However, we intend to adopt static binding.

The second alternative is that, by a form of scope extrusion (see Example 3), the generalized expansion law equates (16) to

$$\tau.(w')(Q\{w'/w\} \mid (P\{w'/w\} \mid R)) \tag{18}$$

where w' has been chosen not free in P, Q or R. This alternative preserves the w-link between P and Q, and preserves its distinction from any w-link possessed by R.

Now let us return to the way we represent the passing of a process parameter in the π-calculus, and we shall see that the effect of (18) is obtained. In place of (16) we write

$$(x)((z)(w)(\overline{x}z.(z.P \mid Q)) \mid x(y).\overline{y}.R) \tag{19}$$

(where $y, z \notin \mathrm{fn}(R)$) which, by expansion, will be equal to

$$\tau.(z)((w)(z.P \mid Q) \mid \overline{z}.R) \tag{20}$$

(The restriction (x) is dropped since $x \notin \mathrm{fn}(P, Q, R)$.) Now, by a change of bound name w to $w' \notin \mathrm{fn}(R)$, followed by (8) in reverse to extend the scope of the restriction (w'), we obtain

$$\tau.(z)(w')(z.P\{w'/w\} \mid Q\{w'/w\} \mid \overline{z}.R) \tag{21}$$

which, by expansion and then discard of the restriction (z), becomes

$$\tau.\tau.(w')(P\{w'/w\} \mid Q\{w'/w\} \mid R) \tag{22}$$

This, but for an extra τ action, is identical with (18).

It may therefore seem that link-passing has all the power of process-passing. This is indeed true, in the presence of recursion; indeed, in a private communication Bent Thomsen has given a translation of a *static*-binding variant of his CHOCS calculus [16] into the π-calculus. In one sense link-passing has greater power, since the link which is passed need not only be the trigger of a process; one may pass – to many different recipients perhaps – the power to interact in different ways with an existing process. In another sense, the power of link-passing is less; for it does not by itself give the ability to *copy* a process, as in $x(p).(p \mid p)$. In particular, the π-calculus without recursion cannot provide the power of general recursion, as the λ-calculus does via the paradoxical combinator $\mathbf{Y}$. We take the view that it is natural to provide recursion explicitly.

Example 7: Values and data structures If the only values with which we wish to compute are drawn from a finite set, say $V = \{v_1, \ldots, v_n\}$, then we can simply designate n names – denoted by $\underline{v_1}, \ldots, \underline{v_n}$ – as constants. (The role of constant names in the theory is dealt with in Section 5.) Clearly the match operator – in its derived form for convenience – can be used to control computation. Consider the case $V = \{\mathrm{t}, \mathrm{f}\}$, the truth values. We set $\underline{\mathrm{t}} = \mathrm{T}$ and $\underline{\mathrm{f}} = \mathrm{F}$. Then a process for simply copying a truth value from one link to another is

$$Copy(y, z) \stackrel{\mathrm{def}}{=} y : [\mathrm{T} \Rightarrow \overline{z}\,\mathrm{T},\ \mathrm{F} \Rightarrow \overline{z}\,\mathrm{F}] \tag{23}$$

(A simpler definition might be $Copy(y, z) \stackrel{\mathrm{def}}{=} y(x).\overline{z}x$, but we are starting a series of definitions which all compute by case-analysis upon the constant or data constructor which is input.) Further, a process $And(x, y, z)$, which produces at z the logical conjunction of the truth values received at x and y, may be defined as follows:

$$And(x, y, z) \stackrel{\mathrm{def}}{=} x : [\mathrm{T} \Rightarrow Copy(y, z),\ \mathrm{F} \Rightarrow \overline{z}\,\mathrm{F}] \tag{24}$$

Now, since we are representing a n-ary boolean function by an agent with $n+1$ link parameters, it is reasonable to extend this to the case $n = 0$. We think of the agents $True(x) \stackrel{\mathrm{def}}{=} \overline{x}\,\mathrm{T}$ and $False(x) \stackrel{\mathrm{def}}{=} \overline{x}\,\mathrm{F}$ as *pointed* values, with x playing the role of pointer. We may then represent application of a function by composition of agents, followed by restriction of the pointer. It is then easy to prove the simple equations which justify the above encoding, such as the following:

$$\begin{array}{rcl} (x)(\mathit{True}(x) \mid \mathit{Copy}(x,y)) & = & \tau.\mathit{True}(y) \\ (x)(y)(\mathit{True}(x) \mid \mathit{False}(y) \mid \mathit{And}(x,y,z)) & = & \tau.\tau.\mathit{False}(z) \\ \cdots & & \cdots \end{array}$$

The matter is different if we wish to compute over an infinite set, for example over the natural numbers Nat. We could choose an infinite family of constants $\{\underline{n} : n \in \mathrm{Nat}\}$, but we cannot write the successor function (for example) as an agent in the form

$$\mathit{Succ}(x,y) \stackrel{\mathrm{def}}{=} x : [\underline{n} \Rightarrow \overline{y}\,\underline{n{+}1}]_{n \in \mathrm{Nat}}$$

because this is an infinite sum, and we want the terms of our calculus to be finite.

To illustrate an alternative method, we shall use the data type of *list structures*, built from a nullary operator 'nil' (the empty list) and a binary operator 'cons'. Any list structure L, say 'cons(cons(nil,nil),nil)', is represented by a pointed value $[\![L]\!](x)$; this is an agent which will emit L piecemeal along the link x. $[\![L]\!]$ is defined as follows, in terms of constant names CONS (for 'cons') and NIL (for 'nil'):

$$[\![\mathrm{nil}]\!](x) \stackrel{\mathrm{def}}{=} \overline{x}\,\mathrm{NIL} \tag{25}$$

$$[\![\mathrm{cons}(L_1,L_2)]\!](x) \stackrel{\mathrm{def}}{=} (y)(z)(\overline{x}\,\mathrm{CONS}.\overline{x}y.\overline{x}z \mid [\![L_1]\!](y) \mid [\![L_2]\!](z)) \tag{26}$$

In the presence of $[\![L]\!](x)$, an agent which possesses or receives the link x thereby possesses or receives the power to explore the list structure L piecemeal, by following pointers. In the case that an agent P *privately* holds the name x of a list structure L, as in the system $(x)(P \mid [\![L]\!](x))$, the transfer of L by P to another agent is therefore a *molecular* action as defined in Example 4. Note particularly that the constituent actions of this molecule may themselves be molecular, since L may have non-trivial sub-structures.

We shall now introduce a few further abbreviations to make the following examples more legible. First we define some composite prefixes:

$$\overline{x}y_1 \cdots y_n \quad \text{means} \quad \overline{x}y_1.\cdots.\overline{x}y_n \tag{27}$$

$$x(y_1)\cdots(y_n) \quad \text{means} \quad x(y_1).\cdots.x(y_n) \tag{28}$$

Thus, if $L = $ cons(cons(nil,nil),nil), then we have

$$[\![L]\!](x) = (y)(z)(\overline{x}\,\mathrm{CONS}yz \mid (v)(w)(\overline{y}\,\mathrm{CONS}vw \mid \overline{v}\,\mathrm{NIL} \mid \overline{w}\,\mathrm{NIL}) \mid \overline{z}\,\mathrm{NIL})$$

Second, we define a more refined form of matching clause:

$$x : [\ldots,\ v(y_1)\cdots(y_n) \Rightarrow P,\ \ldots] \quad \text{means} \quad x : [\ldots,\ v \Rightarrow x(y_1)\cdots(y_n)P,\ \ldots] \tag{29}$$

Thus, when v is received on link x, the names subsequently received on x are bound to $y_1, \ldots, y_n$.

As an example, let us define an agent $\mathit{Equal}(x, y, b)$ which outputs T on b if x and y point to equal structures, F otherwise:

$$\begin{aligned}
\mathit{Equal}(x, y, b) &\stackrel{\mathrm{def}}{=} x : [\mathrm{NIL} \Rightarrow \mathit{Null}(y, b),\ \mathrm{CONS}(x_1)(x_2) \Rightarrow \\
&\qquad\qquad \mathit{Consequal}(x_1, x_2, y, b)] && (30) \\
\mathit{Null}(y, b) &\stackrel{\mathrm{def}}{=} y : [\mathrm{NIL} \Rightarrow \mathit{True}(b),\ \mathrm{CONS} \Rightarrow \mathit{False}(b)] && (31) \\
\mathit{Consequal}(x_1, x_2, y, b) &\stackrel{\mathrm{def}}{=} y : [\mathrm{NIL} \Rightarrow \mathit{False}(b),\ \mathrm{CONS}(y_1)(y_2) \Rightarrow \\
&\qquad (b_1)(b_2)(\mathit{Equal}(x_1, y_1, b_1) \\
&\qquad\qquad \mid \mathit{Equal}(x_2, y_2, b_2) \\
&\qquad\qquad \mid \mathit{And}(b_1, b_2, b))] && (32)
\end{aligned}$$

We hope these simple examples provide convincing evidence for what we shall show rigorously in a later paper, namely that our bare calculus of names is enough to encode a richer calculus in which values of many kinds may be communicated, and in which value computations may be freely mixed with communications. The analogous encoding for CCS [9] relies upon infinite summation; instead, we exploit the power which private links provide to represent complex values as structured agents.

One or two points about the above encoding deserve mention:

- Our pointed values are finite processes; they are ephemeral, in the sense that they may only be analysed once. But other encodings are possible which give permanence to values.

- The encoding has only needed a finite number of constant names: T, F, CONS and NIL. But there are encodings which need no constant names whatever. The trick is to use matching to distinguish *private* names; for example:

$$\begin{aligned}
\mathit{True}(x) &\stackrel{\mathrm{def}}{=} (u)(v)(\overline{x}uvu) \\
\mathit{False}(x) &\stackrel{\mathrm{def}}{=} (u)(v)(\overline{x}uvv)
\end{aligned}$$

  The reader may enjoy re-defining $\mathit{And}(x, y, z)$ to work with these forms.

- We shall now justify the claim made in the introduction that the match form is unnecessary for encoding computation over data types. The control which it provides can, in fact, be achieved by other means. Consider the following agent P, which inputs a truth value (in the encoding we have just presented) on link x, and enters either P_1 or P_2 according to the value:

$$x(u)(v)(w).([w{=}u]P_1 + [w{=}v]P_2)$$

Now let us change the encoding of truth values very slightly:

$$\begin{array}{rcl} \mathit{True}(x) & \stackrel{\text{def}}{=} & (u)(v)(\overline{x}uv.\overline{u}) \\ \mathit{False}(x) & \stackrel{\text{def}}{=} & (u)(v)(\overline{x}uv.\overline{v}) \end{array}$$

Then the agent P can be correspondingly changed to

$$x(u)(v).(u.P_1 + v.P_2)$$

Clearly, both this encoding and the previous one can be extended to deal with any finite set of value constructors or constants.

The attentive reader will have noticed that, in allowing constants to occur free in the equations which define agent identifiers, we have violated the condition on defining equations imposed in Section 2. We justify this violation at the end of Section 5; it is merely part of a conventional treatment of constants.

Example 8: Combinator graph reduction In combinatory logic, *terms* are built from *combinators* by a binary operation called *application*. We let M, N and P range over terms, and we shall consider only the three most basic combinators $\mathbf{S}$, $\mathbf{K}$ and $\mathbf{I}$. The syntax of terms is therefore

$$M \qquad ::= \quad \mathbf{S} \;\mid\; \mathbf{K} \;\mid\; \mathbf{I} \;\mid\; (MN)$$

Application associates to the left, so the term $\mathbf{SK}(MN)\mathbf{S}$ means $(((\mathbf{SK})(MN))\mathbf{S})$. Terms may be *reduced* by the following rules:

$$\begin{array}{rcl} \mathbf{S}MNP & \longrightarrow & MP(NP) \\ \mathbf{K}MN & \longrightarrow & M \\ \mathbf{I}M & \longrightarrow & M \end{array}$$

A *combinator graph* is a graph which represents a term. For every application in the term it contains a node labelled @, with a left and a right child; every other node is labelled by a combinator and has no children. Thus, for the term $\mathbf{S}(\mathbf{K}M)(\mathbf{K}M)N$, either of two graphs shown in Figure 1 will do: The first graph represents *sharing* of two occurrences of the subterm $\mathbf{K}M$.

Combinator graph reduction models term reduction, except that it takes advantage of sharing. It is an important implementation technique for functional programming languages, and computers are being designed to support it by hardware – see for example Goguen *et al* [8]. It will therefore also be important to model the performance of this hardware in a formal calculus, to verify its performance. This presents a tough challenge to the calculus, which must describe not only the mobile structure of the (virtual) processes among themselves, but also their changing allocation to (real) processors. We believe that the π-calculus

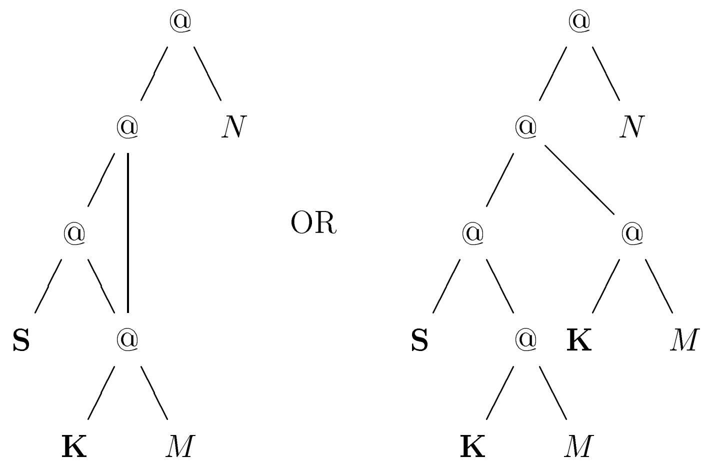


Figure 1: Two combinator graphs for the term $\mathbf{S}(\mathbf{K}M)(\mathbf{K}M)N$

contains the right primitives to meet this challenge. We have given a hint in Example 5 (the Executor) of how allocation to processors may be modelled; the changing virtual structure is combinator graph reduction, to which we now turn.

First, we give the graph reduction rules; see Figure 2. They use auxiliary combinators $\mathbf{S}_1$, $\mathbf{S}_2$, $\mathbf{K}_1$ and $\mathbf{I}_1$ in addition to $\mathbf{S}$, $\mathbf{K}$ and $\mathbf{I}$ (which we shall now call $\mathbf{S}_0$, $\mathbf{K}_0$ and $\mathbf{I}_0$). There is exactly one rule for each combinator, allowing reduction when it occurs as a left child. Note how sharing is introduced by the rule for $\mathbf{S}_2$. Note also that the auxiliary combinators $\mathbf{S}_1, \mathbf{S}_2, \ldots$ appear in the graphs not at the leaves, but as operators of arity one or two.

We shall now illustrate how the *term reduction*

$$\mathbf{S}(\mathbf{K}M)(\mathbf{K}M)N \xrightarrow{(1)} \mathbf{K}MN(\mathbf{K}MN) \xrightarrow{(2)} M(\mathbf{K}MN) \xrightarrow{(3)} MM$$

is modelled by *graph reduction*. We give the graph reduction in Figure 3. Notice that several steps of graph reduction correspond to a single step of term reduction; we show this by numbering the arrows. The *redex* – i.e. the subgraph to be reduced – at each stage is indicated by ringing its @ node. One should note that, just as the subgraph for $(\mathbf{K}M)$ has two parents, so any other node in the graph may have another parent not shown in the diagram (if the whole is a subgraph of a larger system); such nodes, even if they become disconnected during this particular reduction, cannot be discarded altogether. (In passing, note that we have not succeeded in eliminating $\mathbf{I}_1$ entirely; a more sophisticated set of rules can achieve this.)

We can now proceed to model the combinator graphs as (flow graphs of) composite agents. Each combinator $\mathbf{S}_i$, $\mathbf{K}_i$ or $\mathbf{I}_i$ is modelled by an agent with $i{+}1$ parameters; the first i parameters are links to its children, and the last a link to its parent(s). Each combinator repeatedly utters a message, which contains its own identity (a constant name such as s_0) and the names of its children. Here are

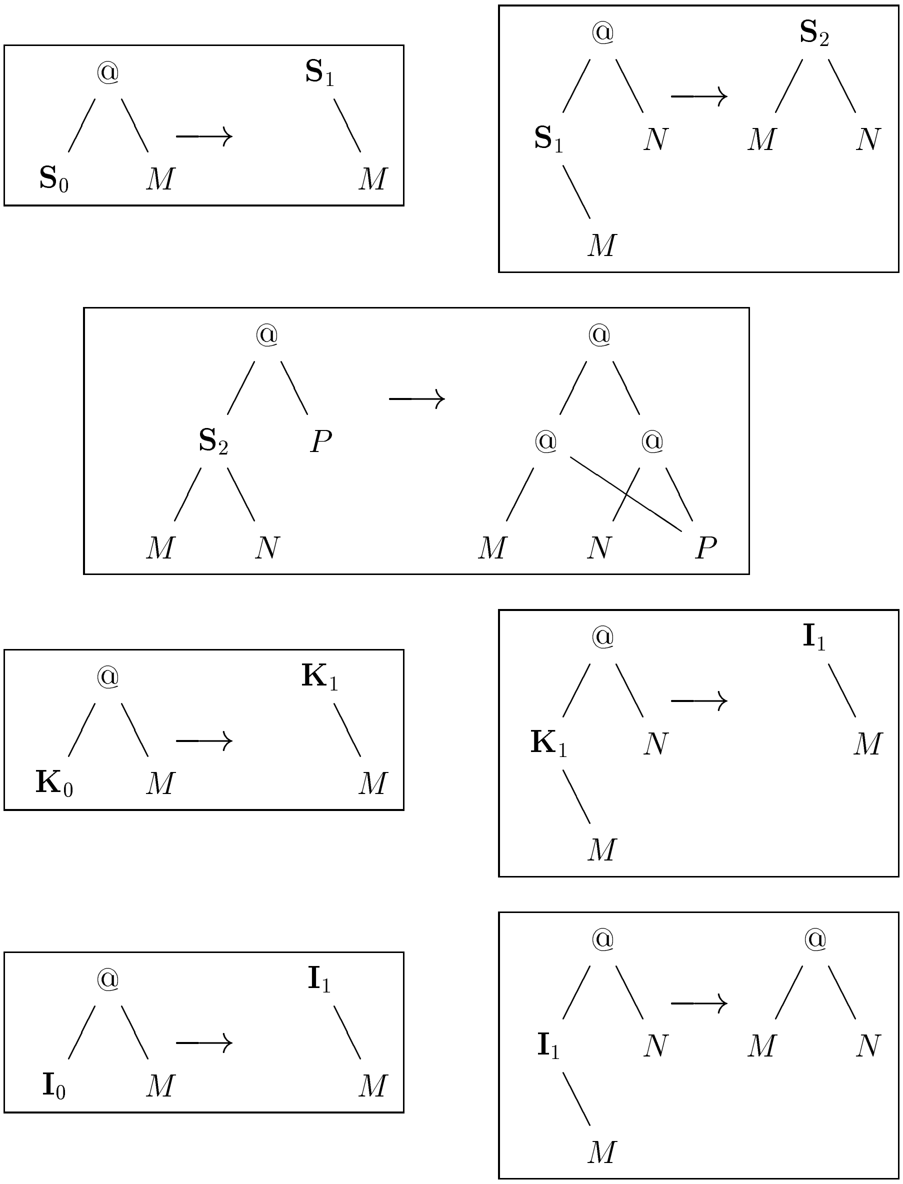


Figure 2: Rules for combinator graph reduction

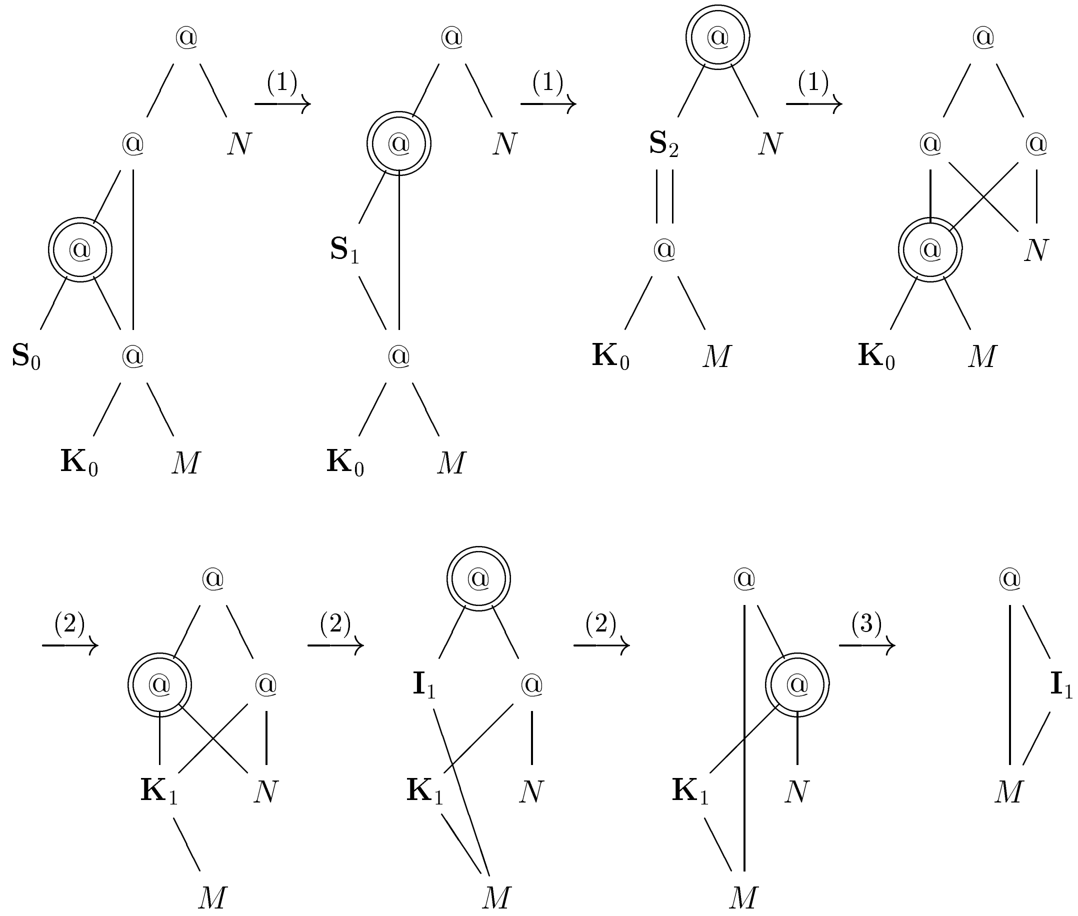


Figure 3: Graph reduction for the term $\mathbf{S}(\mathbf{K}M)(\mathbf{K}M)N$

the definitions for the three $\mathbf{S}$ combinators (the others are completely analogous):

$$\begin{array}{rcl}\mathbf{S}_0(p) & \stackrel{\text{def}}{=} & (w)(\overline{p}w.(\overline{w}\,\mathrm{s}_0 \mid \mathbf{S}_0(p)))\\ \mathbf{S}_1(x,p) & \stackrel{\text{def}}{=} & (w)(\overline{p}w.(\overline{w}\,\mathrm{s}_1 x \mid \mathbf{S}_1(x,p)))\\ \mathbf{S}_2(x,y,p) & \stackrel{\text{def}}{=} & (w)(\overline{p}w.(\overline{w}\,\mathrm{s}_2 xy \mid \mathbf{S}_2(x,y,p)))\end{array}$$

The message sent by $\mathbf{S}_1$, for example, consists of the pointed value $\overline{w}\mathrm{s}_1.\overline{w}x$, formed into a molecular action whose pointer is the private link w. $\mathbf{S}_1$ restores itself after the message, and its next message will have a new private link. The use of molecular actions ensures that different parents do not read parts of the same message.

All that remains is to define the application agent $@(x,y,p)$. It communicates only with its left child, and only when this is a combinator. Each clause of the match, in the definition, corresponds to one of the seven graph reduction rules.

$$\begin{array}{rcl}@(x,y,p) & \stackrel{\text{def}}{=} & x(w).\ w : [\ \mathrm{s}_0 \Rightarrow \mathbf{S}_1(y,p),\\ & & \qquad \mathrm{s}_1(x_1) \Rightarrow \mathbf{S}_2(x_1,y,p),\\ & & \qquad \mathrm{s}_2(x_1)(x_2) \Rightarrow (p_1)(p_2)(@(x_1,y,p_1)\\ & & \qquad\qquad\qquad\qquad \mid @(x_2,y,p_2)\\ & & \qquad\qquad\qquad\qquad \mid @(p_1,p_2,p)),\\ & & \qquad \mathrm{K}_0 \Rightarrow \mathbf{K}_1(y,p),\\ & & \qquad \mathrm{K}_1(x_1) \Rightarrow \mathbf{I}_1(x_1,p),\\ & & \qquad \mathrm{I}_0 \Rightarrow \mathbf{I}_1(y,p),\\ & & \qquad \mathrm{I}_1(x_1) \Rightarrow @(x_1,y,p)\ ]\end{array}$$

With these definitions, the reader can show, without too much difficulty, that (for example) the graph reduction rule for $\mathbf{S}_2$ is captured by the equation

$$\begin{array}{rl} & (p')(\mathbf{S}_2(x,y,p') \mid @(p',z,p)) \mid M(x) \mid N(y) \mid P(z)\\ = & \tau.\tau.\tau.\tau.((p_1)(p_2)(@(x,z,p_1)|@(y,z,p_2)|@(p_1,p_2,p)) \mid M(x) \mid N(y) \mid P(z))\end{array}$$

Finally the reader may like to check that one can avoid using both constants $(\mathrm{s}_0, \mathrm{s}_1, \ldots)$ and the match form in modelling combinator graphs, using alternative encoding as suggested at the end of Example 7.

Example 9: The λ-calculus The encoding of combinator graph reduction (Example 8) has already shown that higher-order functions can be 'handled', in some sense, in the π-calculus. This can be thought of as an encoding of the λ-calculus, since there are natural translations of λ-calculus into combinator algebra; but the encoding is rather indirect. Here we give a much more direct encoding, one in which reduction sequences in the two calculi correspond closely.

It will also show that, to gain the full power of λ-calculus, only a very limited use of recursion is needed – no more than just to achieve *replication* of an agent. More precisely, what we shall encode is a particular reduction strategy for λ-calculus: *lazy* reduction [1]. Theorems which justify this encoding, and also an encoding of *call-by-value* reduction, appear in a separate paper [10].

First, recall the syntax of λ-calculus; its terms $M, N, \ldots \in \Lambda$ have the syntax

$$M \quad ::= \quad x \mid (\lambda x M) \mid (MN)$$

where x ranges over an infinite set $\mathcal{V}$ of variables. We often omit the parentheses around the composite forms, when there is no ambiguity, taking application (MN) to be left-associative. For convenience we assume that $\mathcal{V}$ is a subset of $\mathcal{N}$, with $\mathcal{N} - \mathcal{V}$ infinite, and for this example we take x, y, z to range over $\mathcal{V}$ while u, v, w range over $\mathcal{N} - \mathcal{V}$. Then the lazy reduction relation $\longrightarrow$ over Λ is the smallest relation such that

$$\begin{array}{l} (\lambda x M)N \longrightarrow M\{N/x\} \\ \text{If } M \longrightarrow M' \text{ then } MN \longrightarrow M'N \end{array}$$

This reduction strategy is completely deterministic. Any term M can be written $M_0 M_1 \cdots M_m$ $(m \geq 0)$, where M_0 is not an application. Then $M \longrightarrow M'$ for some M' if and only if $m \geq 1$ and M_0 is $\lambda x N$, and then M' is $N\{M_1/x\}M_2 \cdots M_m$. So for each M there is at most one irreducible term M' such that $M \longrightarrow^* M'$, and moreover M' is either of the form $\lambda x N$ or of the form $x N_1 \cdots N_n$ $(n \geq 0)$.

We shall first encode the *linear* λ-calculus, in which no sub-term of a term may contain more than one free occurrence of x, for any variable x. The paradoxical combinator $\mathbf{Y}$ is thereby excluded; indeed, every reduction sequence terminates in the linear λ-calculus. Correspondingly, we find that we need not call upon recursion for the encoding in the π-calculus.

We shall translate each λ-term M into a map $[\![M]\!]$ from names to agents. To understand the agent $[\![M]\!]u$, where u is any name $(\in \mathcal{N} - \mathcal{V})$, we may think of u *pointing* to the argument sequence appropriate for a particular occurrence of M. More precisely, if M eventually reduces to a λ-abstraction $\lambda x M'$, then the corresponding derivative of $[\![M]\!]u$ will receive along the link u two names: a pointer to M's first argument, and a pointer to the rest of its argument sequence. Thus u represents a list, just as lists are represented in Example 7. Here is the full definition of the encoding function $[\![\ ]\!]$:

$$[\![\lambda x M]\!]u \quad \stackrel{\text{def}}{=} \quad u(x)(v).[\![M]\!]v \tag{33}$$

$$[\![x]\!]u \quad \stackrel{\text{def}}{=} \quad \overline{x}u \tag{34}$$

$$[\![MN]\!]u \quad \stackrel{\text{def}}{=} \quad (v)([\![M]\!]v \mid (x)\overline{v}xu.x(w).[\![N]\!]w) \qquad (x \text{ not free in } N) \tag{35}$$

Note that the variable x occurs free in the translation of the λ-term x; hence in equation (33) x will normally occur free in $[\![M]\!]v$.

The double guarding of $[\![N]\!]$ in equation (35) is the essence of lazy reduction. The first guard, the prefix $\overline{v}xu$, will be activated only when M has reduced to the form $\lambda x M'$ and is ready for its argument; the second guard $x(w)$ will be activated only when M' *calls* N via the name x; only then may the reduction on N commence.

It is illuminating to see how the encoding of a particular example behaves. Consider $(\lambda xx)N$; first, we have

$$[\![\lambda xx]\!]v \quad \equiv \quad v(x)(w).\overline{x}w$$

So, assuming x not free in N:

$$\begin{array}{rcl} [\![(\lambda xx)N]\!]u & \equiv & (v)([\![\lambda xx]\!]v \mid (x)\overline{v}xu.x(w).[\![N]\!]w) \\ & \equiv & (v)(v(x)(w).\overline{x}w \mid (x)\overline{v}xu.x(w).[\![N]\!]w) \\ & = & \tau.(v)(x)(v(w).\overline{x}w \mid \overline{v}u.x(w).[\![N]\!]w) \\ & = & \tau.\tau.(v)(x)(\overline{x}u \mid x(w).[\![N]\!]w) \\ & = & \tau.\tau.\tau.(v)(x)(\mathbf{0} \mid [\![N]\!]u) \\ & = & \tau.\tau.\tau.[\![N]\!]u \end{array}$$

More generally, it is easy to show that

$$[\![(\lambda xM)N]\!]u \quad \approx \quad [\![M\{^N\!/\!_x\}]\!]u \tag{36}$$

where $\approx$ is the weak bisimilarity discussed briefly in Section 5. Moreover the (unique) derivation sequences of both sides are closely related. (They do not keep precisely in step; the left-hand side takes more steps, because it simulates the substitution of N for x in M by making the (only!) occurrence of x in M send its argument list to N.)

The proof of (36) relies strongly on the linearity of M; if M contains x twice then *each* occurrence of x will attempt to send an argument list to N, and this will fail because the agent

$$x(w).[\![N]\!]w$$

(which represents the 'procedure' $[\![N]\!]$ receiving an arbitrary argument list w along x) is consumed by the first call.

In the translation of the full λ-calculus, then, what is needed is *replication*. Let us therefore define, for any action-prefix α, the form

$$\alpha * P \quad \stackrel{\text{def}}{=} \quad \mathbf{fix}X(\alpha.(P \mid X))$$

Here we have used the fixed-point construction $\mathbf{fix}XE$, which stands for a distinguished solution of the agent equation $X = E$. (We could have used such constructions throughout, instead of using agent identifiers and providing them with defining equations; apart from one or two niceties the two approaches amount to the same.) Thus, we have

$$\alpha * P \;\; = \;\; \alpha.(P \mid \alpha * P)$$

which indicates that each 'call' – i.e. each occurrence of the action α – generates a new copy of P. Note that this equation holds even when α is a bound action such as $x(w)$.

This *replicator*, as we shall call it, can now be used to make the only change needed in our translation to accommodate the full λ-calculus, namely to replace equation (35) by

$$[\![MN]\!]u \;\; \stackrel{\text{def}}{=} \;\; (v)([\![M]\!]v \mid (x)\overline{v}xu.(x(w) * [\![N]\!]w)) \qquad (x \text{ not free in } N) \tag{37}$$

Now N may be called more than once from M; each call generates a new replica of N and provides it with a different argument list in place of w. Moreover, with the help of some lemmas about replicators, equation (36) can still be proven, and the close correspondence between the reduction sequence of any M in the λ-calculus and the derivation of its encoding $[\![M]\!]$ is maintained.

Earlier we referred to Bent Thomsen's translation of the static-binding variant of his CHOCS calculus [16] into the π-calculus; in this translation, he independently found that replication is the only use of recursion required.

Abramsky [1] defines a notion of *applicative simulation*, $\lesssim$, for the lazy λ-calculus, and analyses its model theory and proof theory in depth. He actually called it applicative *bi*-simulation, but we prefer to reserve this term for the induced equivalence $\lesssim \cap \gtrsim$, which we shall denote by $\eqsim$. It is therefore natural to ask the relationship between $[\![M]\!]u \approx [\![N]\!]u$ and $M \eqsim N$. It turns out that for closed terms

$$[\![M]\!]u \approx [\![N]\!]u \;\; \text{implies} \;\; M \eqsim N$$

But the converse is false; an example of Ong [13] can be adapted to show this. Intuitively, the reason is that applicative bisimulation only considers the behaviour of a term M when applied to arguments which are λ-terms, while the process $[\![M]\!]u$ inhabits the more unruly environment of arbitrary processes.

Before leaving the λ-calculus we should remark that we have only encoded faithfully one of its reduction strategies, albeit an important one. Much work remains to be done to broaden the connection between the two calculi.

5 Algebraic theory

In our companion paper [11] we give a definition of *strong bisimulation* between agents, and a corresponding equivalence relation of *strong bisimilarity*. We shall use $P \dot{\sim} Q$ to mean that P and Q are strongly bisimilar. Before giving the equational theory of this relation, we point out a subtlety which was of no great concern in CCS, but is important here – namely that strong bisimilarity is not preserved by substitution for free names. For this reason, we shall sometimes refer to strong bisimilarity as *strong ground equivalence*. For example, let x and y be distinct names and consider the equation

$$\overline{x} \mid y \ \dot{\sim}\ \overline{x}.y + y.\overline{x} \tag{38}$$

This holds in our theory, but the substitution of x for y falsifies it; we have

$$\overline{x} \mid x \ \dot{\not\sim}\ \overline{x}.x + x.\overline{x} \tag{39}$$

but on the other hand

$$\overline{x} \mid x \ \dot{\sim}\ \overline{x}.x + x.\overline{x} + \tau \tag{40}$$

This is the price we pay for not distinguishing constants from variables. Later, however, we shall introduce strong (*non-ground*) equivalence $\sim$; it will be simply defined as strong bisimilarity under all substitutions. This relation *is* preserved by substitution, and moreover we shall find the following (more general) equation true:

$$\overline{x}.P \mid y.Q \ \sim\ \overline{x}.(P \mid y.Q) + y.(\overline{x}.P \mid Q) + [x=y]\tau.(P \mid Q) \tag{41}$$

Our equational axioms will use a kind of head normal form. In order to define this form we shall need a new abbreviation:

$$\overline{x}(y).P \ \text{ means }\ (y)\overline{x}y.P \ \text{ if } x \text{ and } y \text{ are distinct} \tag{42}$$

This special case of restriction may be thought of as the simultaneous creation and transmission of a new private name; it is a name which cannot have been 'used before' because it only occurs within P, which only becomes active after the transmission. The importance of this form is that, as our equational theory shows, every use of restriction can be reduced (up to bisimilarity) to this special case.

We now have four kinds of prefix, and we shall allow $\alpha, \beta, \ldots$ to range over them. The syntax of prefixes is

$$\alpha \ ::= \ \tau \ \mid \ x(y) \ \mid \ \overline{x}y \ \mid \ \overline{x}(y)$$

where, of course, the first three are primitive forms and the last is derived.

Definition 1 An agent P is in *head normal form* if it is a sum of prefixes:

$$P \quad \equiv \quad \sum_i \alpha_i.P_i \qquad \square$$

5.1 Strong bisimilarity

We shall now give an equational theory for strong bisimilarity. It will turn out that this theory is complete over *finite* agents, but incomplete over *all* agents (necessarily since $\dot{\sim}$ is not recursively enumerable). We shall state the rules using the standard equality symbol $=$, rather than the symbol $\dot{\sim}$; the reason for this is that, both in this paper and in later work, we shall wish to consider the validity of a rule when $=$ is interpreted by other equivalence relations. For example, Proposition 4 below asserts that several – but not all – of the rules are valid when $=$ stands for strong equivalence, $\sim$.

The reader may wonder why we first axiomatize $\dot{\sim}$, rather than $\sim$, even though the latter is preserved by *all* substitutions (i.e. is a congruence) and is therefore a more natural candidate for the 'equality' of agents. In fact, in Proposition 9 below we do axiomatize $\sim$, but that second axiomatization, as we shall see, depends upon the present one.

We omit the standard rules for an equivalence relation, taking them as given. On the other hand $=$ will not always stand for a congruence relation; in fact the congruence rule **C0** asserts that $=$ is preserved by all operators except the positive prefix, while **C1** asserts a weaker property for positive prefix.

Alpha-conversion

A *From* $P \equiv Q$ *infer* $P = Q$

Congruence

C0 *From* $P = Q$ *infer*

$$\begin{array}{ll} \tau.P = \tau.Q & \overline{x}y.P = \overline{x}y.Q \\ P + R = Q + R & P \mid R = Q \mid R \\ (x)P = (x)Q & [x=y]P = [x=y]Q \end{array}$$

C1 *From* $P\{z/y\} = Q\{z/y\}$, for all names $z \in \mathrm{fn}(P, Q) \cup \{y\}$, *infer*

$$x(y).P = x(y).Q$$

Summation

S0 $P + \mathbf{0} = P$
S1 $P + P = P$
S2 $P + Q = Q + P$
S3 $P + (Q + R) = (P + Q) + R$

Restriction

$$
\begin{array}{ll}
\textbf{R0} & (x)P = P \qquad \text{if } x \notin \mathrm{fn}(P) \\
\textbf{R1} & (x)(y)P = (y)(x)P \\
\textbf{R2} & (x)(P+Q) = (x)P + (x)Q \\
\textbf{R3} & (x)\alpha.P = \alpha.(x)P \qquad \text{if } x \text{ is not in } \alpha \\
\textbf{R4} & (x)\alpha.P = \mathbf{0} \qquad \text{if } x \text{ is the subject of } \alpha
\end{array}
$$

Match

$$
\begin{array}{ll}
\textbf{M0} & [x=y]P = \mathbf{0} \qquad \text{if } x \text{ and } y \text{ are distinct} \\
\textbf{M1} & [x=x]P = P
\end{array}
$$

Expansion

E *Assume* $P = \sum_i \alpha_i.P_i$ and $Q = \sum_j \beta_j.Q_j$, where no α_i (resp. β_j) binds a name free in Q (resp. P); then *infer*

$$P \mid Q \;=\; \sum_i \alpha_i.(P_i \mid Q) + \sum_j \beta_j.(P \mid Q_j) + \sum_{\alpha_i \,\mathrm{comp}\, \beta_j} \tau.R_{ij}$$

where the relation $\alpha_i \,\mathrm{comp}\, \beta_j$ (α_i *complements* β_j) holds in four cases:

1. α_i is $\overline{x}u$ and β_j is $x(v)$; then R_{ij} is $P_i \mid Q_j\{u/v\}$.
2. α_i is $\overline{x}(u)$ and β_j is $x(v)$; then R_{ij} is $(w)(P_i\{w/u\} \mid Q_j\{w/v\})$, where w is not free in $(u)P_i$ or in $(v)Q_j$.
3. α_i is $x(v)$ and β_j is $\overline{x}u$; then R_{ij} is $P_i\{u/v\} \mid Q_j$.
4. α_i is $x(v)$ and β_j is $\overline{x}(u)$; then R_{ij} is $(w)(P_i\{w/v\} \mid Q_j\{w/u\})$, where w is not free in $(v)P_i$ or in $(u)Q_j$.

Identifier

I *From* $A(\tilde{x}) \stackrel{\mathrm{def}}{=} P$ *infer* $A(\tilde{y}) = P\{\tilde{y}/\tilde{x}\}$

We shall call this axiomatic theory **SGE** (for Strong Ground Equivalence); if $P = Q$ can be proved in **SGE** we write

$$\mathbf{SGE} \vdash P = Q$$

or just $\vdash P = Q$ if no ambiguity arises. Note some important points:

1. The last clause of rule **C0**, namely

   $$\textit{From} \quad P = Q \quad \textit{infer} \quad [x=y]P = [x=y]Q$$

   is redundant in the presence of **M0** and **M1**, since any case of it can be deduced from them. But **C0** will be needed when $=$ is interpreted as $\sim$, since **M0** is invalid in that interpretation.

2. Rule **C1** cannot be strengthened to

$$\textit{From} \quad P = Q \quad \textit{infer} \;\; x(y).P = x(y).Q$$

as we can see by considering equation 38 above; we have in fact

$$z(y).(\overline{x} \mid y) \not\dot\sim z(y).(\overline{x}.y + y.\overline{x})$$

because y is a place-holder for any received name, and the received name may be x. Thus the hypothesis of rule **C1** must account for all substitutions; for this purpose, however, only finitely many of them need to be verified.

3. By means of **C0**, **M0** and **M1** all occurrences of a match operator which are not within an input-prefix form can be eliminated from an agent. However, $[y = z]$ cannot be removed from the input-prefix form $x(y).[y = z]P$. (See also the previous point.)

4. In rule **R3** note that α includes in its range the derived prefix $\overline{z}(y)$.

5. In rule **E**, cases 2 and 4 are crucial; they represent the communication of a new *private* name, resulting in a restriction (w) which embraces both sender and receiver in its scope.

The following results are all proved in the companion paper [11], for the definition of $\dot\sim$ which is given there.

Proposition 1 (Soundness) All the laws of **SGE** are valid when $=$ is interpreted as strong bisimilarity, $\dot\sim$. □

A natural constraint upon defined agents is the following:

Definition 2 An agent identifier B is *weakly guarded in* P if every occurrence of B in P is within a prefix form. The agent identifier A is *weakly-guardedly defined* if every agent identifier is weakly guarded in the right-hand side of the definition of A. □

The following now shows the importance of head normal form:

Proposition 2 If every agent identifier is weakly-guardedly defined then, for any agent P, there is a head normal form H such that

$$\mathbf{SGE} \vdash P = H$$

Proof An easy case-analysis upon the structure of P. □

From this, it is a not hard to show that **SGE** is complete for strong bisimilarity of finite agents.

Proposition 3 (Completeness for finite agents) For all finite agents P and Q, if $P \dot{\sim} Q$ then $\mathbf{SGE} \vdash P = Q$.

Proof Given in the companion paper [11]. $\square$

5.2 Strong equivalence

The definition of strong equivalence is now straightforward.

Definition 3 A *substitution* is a function from $\mathcal{N}$ to $\mathcal{N}$. We use σ to stand for a substitution, and postfix substitutions in application. $\{y_i/x_i\}_{1 \leq i \leq n}$ denotes the substitution σ for which $x_i\sigma = y_i$, $1 \leq i \leq n$, and otherwise $x\sigma = x$. $\square$

Definition 4 P and Q are *strongly equivalent*, written $P \sim Q$, if $P\sigma \dot{\sim} Q\sigma$ for all substitutions σ.

Now, when the equality symbol $=$ is interpreted as strong equivalence $\sim$, all the laws of **SGE** hold except for rules **M0** and **E**. The failure of **M0** is clear; equations (38) and (39) indicate why **E** fails. On the other hand, a stronger form of rule **C1** is valid:

C1′ *From* $P = Q$ *infer* $x(y).P = x(y).Q$

It may also be shown that recursive definition preserves $\sim$ (though not $\dot{\sim}$) in an appropriate sense; thus strong equivalence is truly a congruence relation.

Matching can be employed to yield a new form **E′** of the expansion law which is valid for $\sim$:

E′ *Assume* $P = \sum_i \alpha_i.P_i$ and $Q = \sum_j \beta_j.Q_j$, where no α_i (resp. β_j) binds a name free in Q (resp. P); then *infer*

$$P \mid Q \;=\; \sum_i \alpha_i.(P_i \mid Q) + \sum_j \beta_j.(P \mid Q_j) + \sum_{\alpha_i \,\mathrm{opp}\, \beta_j} [x_i = y_j]\tau.R_{ij}$$

where the relation $\alpha_i \,\mathrm{opp}\, \beta_j$ (α_i *opposes* β_j) holds in four cases:

1. α_i is $\overline{x_i}u$ and β_j is $y_j(v)$; then R_{ij} is $P_i \mid Q_j\{u/v\}$.
2. α_i is $\overline{x_i}(u)$ and β_j is $y_j(v)$; then R_{ij} is $(w)(P_i\{w/u\} \mid Q_j\{w/v\})$, where w is not free in $(u)P_i$ or in $(v)Q_j$.
3. α_i is $x_i(v)$ and β_j is $\overline{y_j}u$; then R_{ij} is $P_i\{u/v\} \mid Q_j$.
4. α_i is $x_i(v)$ and β_j is $\overline{y_j}(u)$; then R_{ij} is $(w)(P_i\{w/v\} \mid Q_j\{w/u\})$, where w is not free in $(v)P_i$ or in $(u)Q_j$.

We summarize these facts as follows:

Proposition 4 (Soundness) The laws of $\mathbf{SGE} - \{\mathbf{C1}, \mathbf{M0}, \mathbf{E}\} \cup \{\mathbf{C1}', \mathbf{E}'\}$ are valid when $=$ is interpreted as strong equivalence, $\sim$. □

This system is *not* complete for $\sim$ over finite agents. It may be possible to make it so by adding reasonable laws for matching, but we have not yet succeeded in this. An alternative and perhaps simpler way to axiomatise strong equivalence is given in Proposition 9 below.

In Proposition 5 we give further useful laws of strong equivalence; they are important in the sense that, in exploring alternatives for the semantic definition, we have found them – particularly the last two – a stringent test. It is no exaggeration to say that, without these laws, we would not feel justified in proposing the calculus.

Proposition 5

1. $P \mid \mathbf{0} \sim P$
2. $P \mid Q \sim Q \mid P$
3. $P \mid (Q \mid R) \sim (P \mid Q) \mid R$
4. $(x)(P \mid Q) \sim P \mid (x)Q \quad \text{if } x \notin \text{fn}(P)$

Proof In the companion paper [11]. □

5.3 Recursion

We record here the properties which we would expect of recursive definitions, by analogy with CCS [9]. First, if we transform the right-hand sides of definitions, respecting $\sim$, then the agent defined is the same up to $\sim$. Second, if two agents satisfy the same (recursive) equation, then they are the same up to $\sim$, provided the equation satisfies a standard condition. Both properties fail for $\dot\sim$, strong *bisimilarity*.

In order to state these results, we need a few preliminaries. We assume a set of *schematic identifiers*, each having a nonnegative *arity*. In the following, X and X_i will range over schematic identifiers. An *agent expression* is like an agent, but may contain schematic identifiers in the same way as identifiers; in this section E, F will range over agent expressions.

Definition 5 Let X have arity n, let $\tilde{x} = x_1, \ldots, x_n$ be distinct names, and assume that $\text{fn}(P) \subseteq \{x_1, \ldots, x_n\}$. The *replacement of* $X(\tilde{x})$ *by* P *in* E, written $E\{X(\tilde{x}) := P\}$, means the result of replacing each subterm $X(\tilde{y})$ in E by $P\{\tilde{y}/\tilde{x}\}$. This extends in the obvious way to *simultaneous replacement* of several schematic identifiers, $E\{X_1(\tilde{x}_1) := P_1, \ldots, X_m(\tilde{x}_m) := P_m\}$. □

As an example,

$$\Big(\overline{x}y.X(x,x) + (y)X(x,y)\Big)\{X(u,w){:=}\overline{u}w.\mathbf{0}\} \;\equiv\; \overline{x}y.\overline{x}x.\mathbf{0} + (y)\overline{x}y.\mathbf{0}$$

In what follows, we assume the indexing set I to be either $\{1,\ldots,m\}$ for some $m \geq 1$, or else ω. We write $\widetilde{X}$ for a sequence $X_1, X_2, \ldots$ indexed by I; similarly $\widetilde{P}$, etc. We use i, j to range over I. When a sequence $\widetilde{X}$ of schematic identifiers is implied by context, each with an associated name sequence $\widetilde{x}_i$, then it is convenient to write $E\{X_1(\widetilde{x}_1){:=}P_1, \ldots, X_m(\widetilde{x}_m){:=}P_m\}$ simply as $E(P_1, P_2, \ldots)$, or as $E(\widetilde{P})$. If each P_i is $A_i(\widetilde{x}_i)$, we also write $E(A_1, A_2, \ldots)$ or $E(\widetilde{A})$.

It is natural to define strong equivalence between agent expressions as equivalence under all replacements of schematic identifiers by agents:

Definition 6 Let E and F be two agent expressions containing only the schematic identifiers X_i, each with associated name sequence $\widetilde{x}_i$. Then $E \sim F$ means that

$$E(\widetilde{P}) \sim F(\widetilde{P})$$

for all $\widetilde{P}$ such that $\mathrm{fn}(P_i) \subseteq \widetilde{x}_i$ for each i. □

We can now state our first result, that recursive definition preserves strong equivalence:

Proposition 6 Assume that $\widetilde{E}$ and $\widetilde{F}$ are agent expressions containing only the schematic identifiers X_i, each with associated name sequence $\widetilde{x}_i$. Assume that $\widetilde{A}$ and $\widetilde{B}$ are identifiers such that for each i the arities of A_i, B_i and X_i are equal. Assume that for all i:

$$\begin{array}{rcl} E_i & \sim & F_i \\ A_i(\widetilde{x}_i) & \stackrel{\mathrm{def}}{=} & E_i(\widetilde{A}) \\ B_i(\widetilde{x}_i) & \stackrel{\mathrm{def}}{=} & F_i(\widetilde{B}) \end{array}$$

Then $A_i(\widetilde{x}_i) \sim B_i(\widetilde{x}_i)$ for all i. □

If A is weakly guarded in E then intuitively, from the definition $A \stackrel{\mathrm{def}}{=} E$, we can unfold the behaviour of A uniquely. The next result makes this precise in the general case:

Proposition 7 Assume that $\widetilde{E}$ are agent expressions containing only the schematic identifiers X_i, each with associated name sequence $\widetilde{x}_i$, and that each X_i is weakly guarded in each E_j. Assume that $\widetilde{P}$ and $\widetilde{Q}$ are agents such that $\mathrm{fn}(P_i) \subseteq \widetilde{x}_i$ and $\mathrm{fn}(Q_i) \subseteq \widetilde{x}_i$ for each i. Assume that for all i:

$$\begin{array}{rcl} P_i & \sim & E_i(\widetilde{P}) \\ Q_i & \sim & E_i(\widetilde{Q}) \end{array}$$

Then $P_i \sim Q_i$ for all i. $\Box$

5.4 Distinctions

Having looked at the theories of both strong bisimilarity and strong equivalence, we now address the task of combining them into one.

Definition 7 A *distinction* is a symmetric irreflexive relation between names. We shall let D range over distinctions. A substitution σ *respects* a distinction D if, for all $(x, y) \in D$, $x\sigma \neq y\sigma$. $\Box$

Definition 8 P and Q are *strongly D-equivalent*, written $P \sim_D Q$, if $P\sigma \dot{\sim} Q\sigma$ for all substitutions σ respecting D. $\Box$

Now it is quite natural to record, for certain pairs of agents, the distinction under which they are equivalent; D need involve only the names which are free in the agents. As a simple example, equation (38) can be strengthened to

$$\overline{x} \mid y \ \sim_{\{x,y\}} \ \overline{x}.y + y.\overline{x} \tag{43}$$

Here we have used a natural abbreviation, allowing ourselves to write a set $A \subseteq \mathcal{N}$ when we mean the distinction $A \times A - \mathrm{Id}_{\mathcal{N}}$, which keeps all members of A distinct from each other. (It may turn out that we only need distinctions of this simpler form, but we have not been able to assure ourselves of this.) Clearly, then, we have the two extreme cases

$$\dot{\sim} \ = \sim_{\mathcal{N}} \qquad \text{and} \qquad \sim \ = \sim_{\emptyset}$$

There are two useful operations upon distinctions. First, we define

$$D\backslash x \stackrel{\text{def}}{=} D - (\{x\} \times \mathcal{N} \cup \mathcal{N} \times \{x\})$$

This removes any constraint in D upon the substitution for x. Also, for any set $A \subseteq \mathcal{N}$ of names, we define

$$D{\restriction}A \stackrel{\text{def}}{=} D \cap (A \times A)$$

Proposition 8 The following properties hold for strong equivalence indexed by distinctions:

1. If $D \subseteq D'$ then $P \sim_D Q$ implies $P \sim_{D'} Q$
2. $[x\!=\!y]P \sim_{\{x,y\}} \mathbf{0}$
3. If $P \sim_D Q$ then $(x)P \sim_{D\backslash x} (x)Q$

4. If $P \sim_{D\backslash x} Q$ then $y(x).P \sim_D y(x).Q$
5. If $P \sim_D Q$ and $A = \mathrm{fn}(P, Q)$ then $P \sim_{D\lceil A} Q$ $\square$

Prop 8.1 needs little comment. Prop 8.2 is the proper strengthening of rule **M0** in **SGE**. It also combines pleasantly with the modified expansion law $\mathbf{E}'$; by using it, we can remove summands from an expansion provided we strengthen the distinction. As a very simple example, note first that equation (41):

$$\overline{x}.P \mid y.Q \;\sim_\emptyset\; \overline{x}.(P|y.Q) + y.(\overline{x}.P|Q) + [x\!=\!y]\tau.(P|Q)$$

is an instance of $\mathbf{E}'$; then using Prop 8.2 we can deduce

$$\overline{x}.P \mid y.Q \;\sim_{\{x,y\}}\; \overline{x}.(P|y.Q) + y.(\overline{x}.P|Q) \tag{44}$$

Props 8.3 and 8.4 neatly contrast the two kinds of name-binding. 8.3 indicates that since a restriction (x) *itself* preserves x distinct from other variables, there is no need to enforce the distinction by other means. On the other hand, 8.4 indicates the obligation, in proving equality of positive prefix forms, to allow the bound variable to range over all names. Note that, using 8.3, we can deduce from (44) that

$$(x)(\overline{x}.P \mid y.Q) \;\sim_\emptyset\; (x)(\overline{x}.(P|y.Q) + y.(\overline{x}.P|Q))$$

This is a *full* equivalence, and compared with (41) it does not require the $[x\!=\!y]$ term because the restriction (x) enforces the distinction between x and y. (In passing, note that this expression simplifies further to $y.(x)(\overline{x}.P|Q)$ by **R2**, **R3** and **R4**.) In contrast, using Prop 8.4 with $D = \emptyset$, we deduce from (41)

$$z(y)(\overline{x}.P \mid y.Q) \;\sim_\emptyset\; z(y)(\overline{x}.(P|y.Q) + y.(\overline{x}.P|Q) + [x\!=\!y]\tau.(P|Q)) \tag{45}$$

and the match cannot be discarded.

Prop 8.5 merely asserts that, in an equation $P \sim_D Q$, only the free names in P and Q have any relevance in D.

While Proposition 8 provides useful working laws, we do not need it to obtain a complete axiomatization of strong equivalence indexed by distinctions. This can trivially be done by adding the following law:

D *From* $P\sigma = Q\sigma$, for all σ respecting D, *infer* $P =_D Q$

(A more refined formulation of rule **D** actually confines the hypothesis to finitely many distinct σ.)

Proposition 9 **SGE** $\cup$ $\{$**D**$\}$ is sound, and complete over finite agents, when $=$ and $=_D$ are interpreted as $\dot\sim$ and $\sim_D$ respectively.

Proof Directly from Definition 8. $\square$.

5.5 Weak bisimilarity and equivalence

We now turn briefly to weak bisimilarity. Analogously with CCS, there is a notion of *weak bisimilarity* $\approx$, also called *weak ground equivalence*, which ignores the silent τ actions; it will be treated in a subsequent paper. As in CCS, this equivalence is not preserved by summation; also, like $\dot{\sim}$, it is not preserved by positive prefix (since it is not preserved by substitution). These two defects can be remedied either separately or together; we thus arrive at three further equivalences, the third of which is a congruence:

Definition 9

1. The agents P and Q are (*weakly*) *ground-equal*, written $P \dot{\simeq} Q$, if $P + R \dot{\approx} Q + R$ for all agents R.
2. The agents P and Q are (*weakly*) *equivalent*, written $P \approx Q$, if $P\sigma \dot{\approx} Q\sigma$ for all substitutions σ.
3. The agents P and Q are (*weakly*) *equal*, written $P \simeq Q$, if $P\sigma \dot{\simeq} Q\sigma$ for all substitutions σ. □

(Of course the last two may also be distinction-indexed, by constraining σ.) We shall not pursue these now, but merely point out that the τ laws of CCS are valid for weak ground equality. The τ laws are as follows:

Prefix

P0 $\alpha.\tau.P = \alpha.P$
P1 $P + \tau.P = \tau.P$
P2 $\alpha.(P + \tau.Q) + \alpha.Q = \alpha.(P + \tau.Q)$

Proposition 10 $\mathbf{SGE} \cup \{\mathbf{P0}, \mathbf{P1}, \mathbf{P2}\}$ is sound, when $=$ is interpreted as $\dot{\simeq}$. □

We conjecture that this axiomatization is also complete for finite agents, but the details remain to be checked.

5.6 Constants

We finish with a brief discussion of constants. In our examples in Section 4 we introduced constant names, and we now need to see how they are best handled. The key property of constants, in the general understanding of the term, is that they 'stand for themselves'. In our context, this means simply that they are never instantiated. In particular, we therefore take the liberty – as in (23,24) for example – not to include them among the parameters of an agent identifier

A which uses them in its definition. They could be so included, to meet the condition imposed on defining equations in Section 2; then one would simply include them also in the parameter list of every *use* of A in agent expressions.

More important is that, since constants will never be instantiated, they never run the risk of being identified with one another. Thus, while working in the theory, one may prove equations among agents which use certain constant names, say $\widetilde{v} = \{v_1, \ldots, v_n\}$, and one may take advantage of their 'constanthood' by proving equations indexed by the distinction $D = \widetilde{v}$ (or, more explicitly, $\widetilde{v} \times \widetilde{v} - \mathrm{Id}_{\mathcal{N}}$). In this working, one may by convention choose to omit the index D from equations. Later, one may wish to abstract from the particular choice of constant names. But this is the essence of Proposition 8.3 (or its analogue for $\simeq$); from any D-indexed equation $P =_D Q$, with $D = \widetilde{v}$, one can infer

$$(\widetilde{v})P =_{\emptyset} (\widetilde{v})Q$$

Thus the calculus reflects the idea that the difference between constants and variables should not be sharply drawn.

6 Conclusion

An algebraic process calculus with mobility has been long in maturing. In 1979, before CCS was published, one of us (Milner) discussed with Mogens Nielsen at Aarhus the possibility of including such a notion at the outset, but we failed to see how to do it. It was not until the paper by Engberg and Nielsen [6] that the possibility was established; their semantic rules represent our starting point. In two ways it has been fortunate that the various process algebras – for example CSP [7], ACP [3] and CCS [9] – did not include mobility: First, they were thereby simpler, and yet presented many problems which were better tackled in a simpler setting; second, the situations in which mobility is needed have become more sharply defined, and therefore the need more sharply felt, through experimental use of these algebras.

There have been a number of formalisms which allow mobility, but have not developed its algebraic theory. The first was Hewitt's Actor formalism. Hewitt's ideas on the changing configuration among actors were developed in the early 1970s; a semantic treatment is given by Clinger in his PhD thesis [5]. More recently, Kennaway and Sleep invented their LNET and DyNe formalisms specifically to describe parallel graph reduction processes, such as we present in Section 4, in the context of a project to design a parallel processor [15]. Also Astesiano and Zucca [2] have extended CCS to include parametric channels.

Engberg and Nielsen [6] did not publish their report, and it has not received due attention, probably because its treatment of constants, variables and names is somewhat difficult. Many features of the π-calculus are due to them, in particular: the replacement of CCS relabelling by syntactic substitution (crucial for formulation of the semantic rules); the semantic treatment of scope extrusion; the extension of the definition of bisimulation to account for name parameters; the definition of strong bisimilarity (which they call simply "strong equivalence"); and the soundness of most algebraic laws. We made many failed attempts to depart from their formulation. Our contribution has been: to remove all discrimination among constant names, variable names and values, yielding a more basic calculus; to discriminate between ground and non-ground equivalence (needed to replace the constant-variable discrimination); to strengthen the algebraic laws – in particular the expansion law – in order to achieve complete equational theories; to encode value-computations in the calculus in a tractable way (with the help of a new match construct); and to provide rather simple encodings of functional calculi – the λ-calculus and combinatory algebra.

References

[1] Abramsky, S., *The Lazy Lambda Calculus*, to appear in **Declarative Programming**, ed. D. Turner, Addison Wesley, 1988.

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

[3] Bergstra, J.A. and Klop, J-W., *Algebra of Communicating Processes with Abstraction*, Journal of Theor. Comp. Science, Vol 33, pp77–121, 1985.

[4] Boudol, G., *Towards a Lambda-Calculus for Concurrent and Communicating Systems*, INRIA Sophia-Antipolis, private communication, 1988.

[5] Clinger, W.D., *Foundations of Actor Semantics*, AI-TR-633, MIT Artificial Intelligence Laboratory, 1981.

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

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

[8] Leinwand, S., Goguen, J.A. and Winkler, T., *Cell and Ensemble Architecture for the Rewrite Rule Machine*, Proc. International Conference on Fifth Generation Computing Systems, ICOT, pp869–878, 1988.

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

[10] Milner, R., *Functions as Processes*, Research Report 1154, INRIA, 1990. Also in abbreviated form in Proc. ICALP '90.

[11] Milner, R., Parrow, J.G. and Walker, D.J., *A Calculus of Mobile Processes, Part II*, Report ECS-LFCS-89-86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.

[12] Nielson, F., *The Typed λ-calculus with First-class Processes*, Report ID-TR:1988-43, Inst. for Datateknik, Tekniske Hojskole, Lyngby, Denmark, 1988.

[13] Ong, C-H.L., *Fully Abstract Models of the Lazy Lambda Calculus*, Proc 29th Symposium on Foundations of Computer Science, pp368–376, 1988.

[14] Reisig, W., **Petri Nets**, EATCS Monographs on Theoretical Computer Science, ed. W.Brauer, G.Rozenberg, A.Salomaa, Springer Verlag, 1983.

[15] Sleep, M.R. and Kennaway, J.R., *The Zero Assignment Parallel Processor (ZAPP) Project*, in **The Distributed Computing Systems Programme**, ed. D.A.Duce, Peter Peregrinus Ltd., pp250–269, 1984.

[16] Thomsen, B., *A Calculus of Higher-order Communicating Systems*, Proc POPL Conference, 1989.