Functions as Processes [*]

Robin Milner
Dept. of Computer Science, University of Edinburgh
Edinburgh EH9 3JZ, Scotland

Abstract

This paper exhibits accurate encodings of the λ-calculus in the π-calculus. The former is canonical for calculation with functions, while the latter is a recent step [15] towards a canonical treatment of concurrent processes. With quite simple encodings, two λ-calculus reduction strategies are simulated very closely; each reduction in λ-calculus is mimicked by a short sequence of reductions in π-calculus. Abramsky's precongruence of *applicative simulation* [1] over λ-calculus is compared with that induced by the encoding of the lazy λ-calculus into π-calculus; a similar comparison is made for call-by-value λ-calculus.

The part of π-calculus which is needed for the encoding is formulated in a new way, inspired by Berry's and Boudol's Chemical Abstract Machine [5].

1 Introduction

This paper shows how the operational model of pure functional programming, where concurrency and state are only implicit, fits comfortably inside an operational model of concurrent processes where concurrency and state are dominant features. More precisely, we exhibit accurate encodings of the λ-calculus in the π-calculus. The former is canonical for calculation with functions, while the latter is a recent step [15] towards a canonical treatment of concurrent processes. We show that, with quite simple encodings, at least two λ-calculus reduction strategies can be simulated very closely; each reduction in λ-calculus is mimicked by a short sequence of reductions in π-calculus. For the encoding of lazy λ-calculus, we compare Abramsky's precongruence over λ-terms known as *applicative simulation* [1] with that which is induced by the encoding; we also make a similar comparison for the call-by-value λ-calculus.

Process calculi of an algebraic nature have been studied and used for about a decade. Most of them lack two related features [3, 11, 13]; the ability to treated *processes* as values which may be transmitted from one (perhaps higher-order) process to another, and the ability to treat *inter-process links* as transmissible values. The lack of these features has been both an advantage and a limitation. An advantage, because the algebraic theory has been simpler – or at least has been easier to develop – than seemed possible with either feature present; a limitation, because without those features certain phenomena can be modelled at best indirectly.

[*]This research was partly done during sabbatical leave from Edinburgh University, on a four-month visit to INRIA at Sophia Antipolis, France. I am grateful both to Edinburgh University and to the French Ministry of Technology for their support.

Recently, several generalisations of process calculus to embrace processes-as-values have been proposed; examples are by Boudol [6], F.Nielsen [16] and Thomsen [19]. Explicitly or implicitly, they have enabled embedding of the λ-calculus; this is a useful measure of power. Indeed, these proposals share with functional calculi the idea that one can *encapsulate as a value* an agent which can be instantiated, or excited, or applied to yield a computation; we therefore say that they follow the *functional* paradigm.

On the other hand, there have been proposals closer to the notion of links-as-values. In one approach, that of Astesiano and Zucca [2], the effect was achieved indirectly by allowing links to be *value-dependent* rather than to be themselves values. By maintaining this separation between links and values the authors were indeed able to present their model as a generalisation of CCS, retaining its equational laws. There have also been more direct approaches, allowing links to be themselves values; of these, the work of Engberg and M.Nielsen [9] was until recently the most theoretically developed, retaining a significant amount of algebraic theory.

The π-calculus, first presented in [15], builds upon this last work. (Engberg and Nielsen did not publish their report, and it has not received due attention – probably because its treatment of constants, variables and names is somewhat complex.) In common with other "links-as-values" approaches, it replaces the functional paradigm by a significantly different paradigm. In the latter, which we may call the *object*[1] paradigm, that which is transmitted and bound is never an agent, but rather *access* to an agent. It is only as a special case that one agent may have *sole* access to another. The object paradigm is hardly new, but there has never been a canonical encapsulation of it, in the way that λ-calculus encapsulates the function paradigm. But there have been signs of its power; for example, following an early ideas of Hewitt [8], even a humble data value can be modelled as an independent agent – one which remains constant (i.e. invariant under access.)

Both paradigms seem equally basic and significant. Without arguing naïvely *in favour* of the object paradigm, our aim in the π-calculus has been to present it in undiluted form. Since higher-order (agent) parameters are sacrificed, a succinct translation between the paradigms is by no means preordained; all that we know is that *some* translation from the function to the object paradigm must exist, since functional languages are successfully implemented on machines which – with their arithmetic units, programs and registers – can be seen just as assemblies of objects.

Another theme of the present paper is a new way of formulating the π-calculus, or at least the fragment of π-calculus that we need for encoding λ-calculus, inspired by Berry's and Boudol's Chemical Abstract Machine [5]. Their analogy of an aggregate of processes moving and interacting within a *solution* has probably occurred vaguely to many people, but Berry and Boudol have made the analogy technically robust. We reflect their intuition here by means of axioms for a structural congruence relation over process terms; this yields a welcome simplicity in presenting the reduction rules.

The paper is self-contained as far as its main purpose, encoding λ-calculus, is concerned; no knowledge of our original presentation of π-calculus [15] is needed. The proofs of the results given here can be found in a technical report [14], which also establishes complete consistency between this presentation and the original.

[1]The connotation here is with object-oriented programming – in particular, the idea of objects with independent state interacting with one another.

2 The π-calculus: Terms

We presuppose an infinite set $\mathcal{N}$ of *names*. We shall use $x, y, z, \ldots$ and sometimes other small letters to range over $\mathcal{N}$.

The π-calculus consists of a set $\mathcal{P}$ of *terms*, sometimes called *agents*, which intuitively stand for processes. We shall use P, Q, R to range over $\mathcal{P}$. We shall write $P\{y/x\}$ to mean the result of replacing free occurrences of x by y in P, with change of bound names where necessary, as usual. (There will be two ways of binding names.)

The first class of terms consists of *guarded terms* $g.P$, where P is a term and g is a *guard*; guards g have the form

$$g \quad ::= \quad \overline{x}y \quad | \quad x(y)$$

Informally, $\overline{x}y.P$ means 'send the name y along the link named x, and then enact P'; on the other hand, $x(y).P$ means 'receive any name z along the link named x, and then enact $P\{z/y\}$'. Thus the guard $x(y)$ is like the λ-prefix λy in that it binds y; it is unlike λy in that every name $x \in \mathcal{N}$ is a binder like λ, but that only names (not terms) may replace bound names.

The second class of terms express concurrent behaviour. The principal form is *composition* $P|Q$, which, informally speaking, enacts P and Q concurrently allowing them to interact via shared links (i.e. shared names). Interaction can occur in the case

$$x(y).P \mid \overline{x}z.Q \tag{1}$$

and we expect the result of interaction to be $P\{z/y\}|Q$. This differs from β-reduction $(\lambda x M)N \rightarrow M\{N/x\}$ in one essential: the 'sender' $\overline{x}z.Q$ pursues an independent future (as Q) after the interaction, while in β-reduction the future behaviour of the argument N is controlled by M via the variable x (in a way which varies from one reduction strategy to another). This exactly reflects the crucial difference, mentioned in the introduction, between the *function* and *object* paradigms.

Allied to composition is *replication*, $!P$; roughly, it stands for $P|P|\cdots$, as many concurrent instances of P as you like. Also allied is *inaction*, the degenerate composition of no processes, denoted by $\mathbf{0}$.

The third class of terms has only one form: the *restriction* $(x)P$. It confines the use of x as a link to within P. Thus, no *intraaction*, i.e. interaction between components, can occur in

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

On the other hand, if (x) is applied to (1) then no *inter*action at x can occur between this term and any terms which may surround it; one therefore expects the equation

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

for some congruence relation $\approx$.

To summarise: for this paper, the syntax of $\mathcal{P}$ is

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

There are a few differences from the π-calculus given in [15]. Only one is a novelty: the presence of $!P$. Replication[2] replaces recursion; for many purposes replication does

[2]To the original rules [15] of π-calculus, add the following (misstated in [14], footnote 2):

$$\frac{P|!P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

the job of recursion (perhaps even for all useful purposes), and is simpler to handle theoretically. Otherwise, we are dealing with a sub-calculus. We have omitted $\tau.P$ (silent guard), $[x{=}y]P$ (matching) and $P+Q$ (summation). The first two present no difficulty for the present formulation, nor does summation in the limited form $\sum_i g_i.P_i$ (sum of guarded terms). It is open how best to handle full summation in the present formulation; but see the method adopted by Berry and Boudol in the Chemical Abstract Machine [5].

Some technical details and terminology:

- There are two forms of binding: $x(y)$ and (y). Note that x is free in $x(y).P$. We use fn(P) for the free names of P, bn(P) for the bound names of P, and n(P) for all names occurring in P.
- We call x the *subject* and y the *object* of the guards $x(y)$ and $\overline{x}y$.
- We say that an occurrence of a term Q in P is *guarded* if it occurs within any guarded term in P, otherwise it is *unguarded.*
- We shall often use $\tilde{x}$ to mean a sequence $x_1, \ldots, x_n$ of names; similarly $\tilde{P}$ for a sequence of terms. Without risk of confusion, we also treat $\tilde{x}$ sometimes as a set. We also write $(\tilde{x})$ for the multiple restriction $(x_1)\cdots(x_n)$.
- We shall use several convenient abbreviations:
  - We shall often omit '.$\mathbf{0}$' in an agent, and write for example $\overline{x}y$ instead of $\overline{x}y.\mathbf{0}$.
  - We shall elide several guards with the same subject, for example $x(y)(z)$ means $x(y).x(z)$ and $\overline{x}yz$ means $\overline{x}y.\overline{x}z$.
  - The agent $(y)\overline{x}y.P$ can be thought of as simultaneously creating and sending a new private name, when $x{\neq}y$; we abbreviate it to $\overline{x}(y).P$.

  So with all these abbreviations we shall be able to write agents like $x(y)(z).\overline{y}z(x)$, meaning $x(y).x(z).\overline{y}z.(x)\overline{y}x.\mathbf{0}$.

3 The π-calculus: Equations and Reductions

Our operational intuition is simple: if term R contains two unguarded subterms $x(y).P$ and $\overline{x}z.Q$, and each restriction (x) contains both or neither (so that x means the same for both subterms), then they can interact; this interaction yields a reduction $R \rightarrow R'$. A few examples:

- Let R be $x(y).P \mid \overline{x}z_1.Q_1 \mid \overline{x}z_2.Q_2$. There are two reductions

$$\begin{aligned} R &\rightarrow P\{z_1/y\} \mid Q_1 \mid \overline{x}z_2.Q_2 \\ R &\rightarrow P\{z_2/y\} \mid \overline{x}z_1.Q_1 \mid Q_2 \end{aligned}$$

- Let R be $w(x).(x(y).P \mid \overline{x}z.Q)$. There is no reduction; the subterms are guarded.
- Let R be $x(y).P \mid (x)(\overline{x}z.Q)$. There is no reduction.

- Let R be $x(y).P \mid (z)(\overline{x}z.Q)$. Assuming z not free in P there is a reduction

$$R \;\rightarrow\; (z)(P\{^z/_y\} \mid Q)$$

  We call this phenomenon *extrusion* (of the scope of a restriction); the name z is private to $(z)\overline{x}z.Q$, but its transmission has enlarged its scope to embrace the recipient.

To simplify the form of the reduction rules, we first define a structural congruence relation over terms. This approach is inspired by Boudol and Berry [5], though our formulation differs from theirs. The idea is that one should separate the laws which govern the neighbourhood relation among processes from the rules which specify their interaction.

3.1 Definition *Structural congruence*, written $\equiv$, is the smallest congruence over $\mathcal{P}$ satisfying these equations:

(1) $P \equiv Q$ whenever P is alpha-convertible to Q

(2) $P \mid \mathbf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$

(3) $!P \equiv P \mid !P$

(4) $(x)\mathbf{0} \equiv \mathbf{0}$, $(x)(y)P \equiv (y)(x)P$

(5) $(x)(P \mid Q) \equiv P \mid (x)Q$ if x not free in P ■

A few facts are easily seen:

- Using the equations, all unguarded restrictions can be moved outermost. Note particularly: $!(x)P \equiv (x)P \mid !(x)P \equiv (x)(P \mid !(x)P)$.
- The interaction condition mentioned at the start of this section is invariant under $\equiv$.
- Using the equations, any two potential interactors $x(y).P$, $\overline{x}z.Q$ can be brought together as $x(y).P \mid \overline{x}z.Q$, but possibly with alpha-conversion; for example, if z is free in P then

$$x(y).P \mid (z)(\overline{x}z.Q) \;\equiv\; (z')(x(y).P \mid \overline{x}z'.Q\{^{z'}/_z\})$$

  where z' is new.

3.2 Definition *Reduction*, written $\rightarrow$, is the smallest relation satisfying the following rules:

$$\text{COM}: \quad x(y).P \mid \overline{x}z.Q \rightarrow P\{^z/_y\} \mid Q$$

$$\text{PAR}: \quad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$$

$$\text{RES}: \quad \frac{P \rightarrow P'}{(y)P \rightarrow (y)P'}$$

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

■

It is worth noticing that, just because of equation (3) in 3.1 for replication, structural congruence may be hard to determine (perhaps even undecidable), and this may cause some alarm in seeing the rule STRUCT, since we certainly want $\rightarrow$ to be computable. But we are saved by the invariance of the interaction condition under $\equiv$, noted above; the interactions possible in a given term are quite manifest. In fact:

3.3 Proposition A finite set Red(P) of agents can be recursively computed from P, such that $P \rightarrow P'$ if and only if $P' \equiv P''$ for some $P'' \in$ Red(P). ■

The present formulation of π-calculus differs strikingly from that in [15]. The essential difference is in the use of structural congruence. This is inspired by the Chemical Abstract Machine of Berry and Boudol [5]. Their insight is that the rules of structured operational semantics, as used in [13] or [15] for example, treat in the same way two concepts which it is worth separating: the *physical structure* of a family of concurrent agents and their *interaction*. Of course, one advantage of avoiding the imposition of structural congruence equations as axioms is that, once a *behavioural* congruence is defined – as in [13] – it turns out that the structural equations are obtained as *theorems*, and this gives added confidence. But then these equations are not kept clearly distinct from other theorems which only hold because of the particular way in which observation is characterised in the behavioural congruence. (For example, observation congruence in [13] is based upon the idea of a *sequential observer*, and yields equations which do not hold in a model which respects causality.) By proposing the structural laws as axioms, one makes the distinction and achieves some simplicity at the same time (at least, when summation is not considered); one also offers the challenge to find an interesting behavioural congruence which fails to satisfy the axioms (strongly suspecting that there is none!)

For the next section we shall need the following:

3.4 Definition P is *r-determinate* if, whenever $P \rightarrow^* Q$ and also $Q \rightarrow Q_1$ and $Q \rightarrow Q_2$, then $Q_1 \equiv Q_2$. Also, P *converges* to Q, written $P \downarrow Q$, if $P \rightarrow^* Q \not\rightarrow$; we write $P \downarrow$ to mean P converges to some Q, and $P \uparrow$ otherwise. ■

4 The lazy λ-calculus

Let the set of *Variables*, $\mathcal{X}$, be an infinite and co-infinite subset of $\mathcal{N}$. For this section, we shall let x, y, z range over $\mathcal{X}$, and u, v, w range over $\mathcal{N} - \mathcal{X}$. Our encoding of λ-calculus into π-calculus will be all the simpler because we treat a variable x of λ-calculus also as a name of π-calculus.

We shall use L, M, N to range over the *terms* $\mathcal{L}$ of λ-calculus, which are defined as usual by:

$$M \quad ::= \quad x \quad | \quad \lambda x M \quad | \quad MN$$

the last two forms being *abstraction* and *application*. The *free variables* fv(M) of a term M are defined in the usual way. $\{^N/x\}$ means substitution as usual. We shall frequently need the the sequential composition of several substitutions (note: *not* simultaneous substitution); so, understanding the members of $\tilde{x}$ to be distinct, we introduce the abbreviation

$$\{\widetilde{N}/\widetilde{x}\} \quad \text{stands for} \quad \{N_1/x_1\}\cdots\{N_k/x_k\}$$

We shall use the standard terms

$$\begin{array}{llll} \mathbf{I} \overset{\text{def}}{=} \lambda x x & & \mathbf{Y} \overset{\text{def}}{=} \lambda f(\lambda x\, f(xx))(\lambda x\, f(xx)) \\ \mathbf{K} \overset{\text{def}}{=} \lambda x \lambda y x & & \mathbf{\Omega} \overset{\text{def}}{=} (\lambda x\, xx)(\lambda x\, xx) \end{array}$$

There are many reduction relations $\rightarrow$, all of which satisfy the rule

$$\beta : \quad (\lambda x M)N \rightarrow M\{N/x\}$$

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:

4.1 Definition The *lazy reduction relation* $\rightarrow$ over $\mathcal{L}$ is the smallest which satisfies β, together with the rule

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

■

With the usual convention that LMN means $(LM)N$, this implies that in any term M, writing it as

$$M \equiv M_0 M_1 M_2 \cdots M_n \qquad (n \geq 0)$$

where M_0 is not an application, the only reduction possible is when $n \geq 1$ and $M_0 \equiv \lambda x N$, and then the reductum is

$$N\{M_1/x\}M_2 \cdots M_n$$

Thus $\rightarrow$ is r-determinate – i.e. every M is r-determinate; given M, there is at most one M' for which $M \rightarrow M'$. We write $\rightarrow^+$ for the transitive closure of $\rightarrow$, and $\rightarrow^*$ for the transitive reflexive closure.

4.2 Definition M *converges* to M', written $M{\downarrow}M'$, if $M \rightarrow^* M' \not\rightarrow$; also, $M{\downarrow}$ means that M converges to some M'. ■

If $M \downarrow M'$, then M' can only be an abstraction $\lambda x N$ or else of the form $xN_1 \cdots N_n$ $(n \geq 0)$. Writing $\mathcal{L}^0$ for the *closed* terms, if $M \in \mathcal{L}^0$ then $M' \in \mathcal{L}^0$ also, so M' must be an abstraction.

Abramsky [1] defines an important preorder $\lesssim$, which we shall call *applicative simulation*, as follows:

4.3 Definition Let $L, M \in \mathcal{L}^0$. Then $L \lesssim M$ if, for all sequences $\widetilde{N}$ in $\mathcal{L}^0$:

$$L\widetilde{N}{\downarrow} \text{ implies } M\widetilde{N}{\downarrow}$$

If $L, M \in \mathcal{L}$ with free variables $\widetilde{x}$, we define $L \lesssim M$ to mean that $L\{\widetilde{N}/\widetilde{x}\} \lesssim M\{\widetilde{N}/\widetilde{x}\}$ for all $\widetilde{N}$ in $\mathcal{L}^0$. ■

Abramsky continues to study both the model theory and the proof theory of $\lesssim$ and of applicative *bisimulation*, $\eqsim$ (i.e. $\lesssim \cap \gtrsim$). We need very little of this here, but should remark that it firmly establishes the importance both of lazy reduction and of these relations, and hence provides a natural point at which process calculus may try to make contact with λ-calculus. We recall from [1] that $(\lambda x M)N \eqsim M\{^N/x\}$, and that $\lesssim$ is a precongruence. The latter follows from the result that, defining a *context* $\mathcal{C}[\_]$ to be a term with a single hole,

$$M \lesssim N \text{ iff, for every closed context } \mathcal{C}[\_], \mathcal{C}[M]\downarrow \text{ implies } \mathcal{C}[N]\downarrow$$

We now turn to the encoding of $\mathcal{L}$ into $\mathcal{P}$. Perhaps one hardly expects to find a more basic calculus than the λ-calculus. All the same, it takes as primitive the remarkably complex operation of substitution (of terms for variables). Two important means have been found to break this operation into smaller parts. In combinatory algebra [7], Curry found combinators which progressively distribute the argument of an abstraction $\lambda x M$ to those parts of the body M which will use it (thus, in fact, eliminating variables altogether). On the other hand, implementations of functions and procedures in programming languages have traditionally used the notion of *environment*, a map from variables to terms; thus, instead of executing $M\{^N/x\}$ one executes M itself in an environment which binds N to the variable x. The encoding which follows can be seen as a formalisation of the latter idea.

Each $M \in \mathcal{L}$ is encoded as $[\![M]\!]$, a map from names to $\mathcal{P}$. Thus $[\![M]\!]u$ is a term of π-calculus, and will have free names given by

$$\text{fn}([\![M]\!]u) \; = \; \text{fv}(M) \cup \{u\}$$

The name u is the link along which $[\![M]\!]$ 'receives' its arguments.

Now, suppose that M will itself be used in place of an argument represented by the variable x. Each time M is 'called', via x, it must be told by the caller where to receive its own arguments. (In more familiar terminology, it must be given a *pointer* to its arguments.) The 'environment entry' binding x to M is therefore the π-term

$$[\![x := M]\!] \overset{\text{def}}{=} !x(w).[\![M]\!]w$$

In passing, note particularly the replication. This is not needed if M will be called at most once; therefore the *linear* λ-calculus, in which each variable x must occur exactly once in its scope, may be encoded in the fragment of π-calculus without replication. The link with Girard's 'of course' connective '!' of linear logic [10] should be explored; his notation has been chosen here deliberately.

How does $[\![\lambda x M]\!]u$ receive its arguments? Along u it receives (as x) the name of its first argument, and also the name of a link where the rest will be transmitted. This explains the first line of our encoding, which we now give in full:

$$\begin{aligned}
[\![\lambda x M]\!]u &\overset{\text{def}}{=} u(x)(v).[\![M]\!]v \\
[\![x]\!]u &\overset{\text{def}}{=} \overline{x}u \\
[\![MN]\!]u &\overset{\text{def}}{=} (v)([\![M]\!]v \mid \overline{v}(x)u.[\![x := N]\!]) \\
& \qquad (x \text{ not free in } N)
\end{aligned}$$

Let us look at the reduction of a simple example, in which we assume x not free in N (recall the abbreviations listed at the end of Section 2):

$$\begin{array}{rll} [\![(\lambda x x)N]\!]u & \equiv & (v)(v(x)(w).\overline{x}w \mid \overline{v}(x)u.[\![x:=N]\!]) \\ & \rightarrow & (v)(x)(v(w).\overline{x}w \mid \overline{v}u.[\![x:=N]\!]) \qquad (2) \\ & \rightarrow & (x)(\overline{x}u \mid [\![x:=N]\!]) \qquad (3) \\ & \equiv & (x)(\overline{x}u \mid !x(w).[\![N]\!]w) \\ & \rightarrow & [\![N]\!]u \mid (x)[\![x:=N]\!] \qquad (4) \\ & \sim & [\![N]\!]u \qquad (5) \end{array}$$

The following remarks will help in reading the above calculation:

- In obtaining (2), recalling that $\overline{v}(x).Q$ means $(x)\overline{v}x.Q$, equation (5) of Definition 3.1 must first be used to allow COM to be applied.
- The restriction (v) is dropped in (3) because v no longer occurs. Formally, if $v \notin \text{fn}(R)$ then

$$(v)R \equiv (v)(R \mid \mathbf{0}) \equiv R \mid (v)\mathbf{0} \equiv R \mid \mathbf{0} \equiv R$$

- In (4), (x) has been moved inwards, since $x \notin \text{fn}([\![N]\!]u)$.
- The last step, to (5), is the only one which goes beyond $\equiv$; it is a simple case of *strong bisimilarity* – see [15] – and represents the garbage-collection of an environment entry $[\![x:=N]\!]$ which cannot be used further (since the subject x of its first action is restricted).

The correspondence between the reduction of $[\![M]\!]$ in the π-calculus and that of M in the λ-calculus is very close. Each step in λ-calculus is matched – in a fully determinate way – by a short sequence of steps in π-calculus. The essential difference, as was mentioned earlier, is that substitutions $\{M/x\}$, which are actually performed upon λ-terms, are represented in $\mathcal{P}$ by what we have called environment entries $[\![x:=M]\!]$ which are agents in their own right.

Let us introduce the abbreviation

$$[\![\widetilde{x}:=\widetilde{N}]\!] \quad \text{stands for} \quad [\![x_1:=N_1]\!] \mid \cdots \mid [\![x_k:=N_k]\!]$$

for the composition of several environment entries. Then the match between the reduction of *closed* λ-terms and the corresponding π-terms leads to the following theorem:

4.4 Theorem (Lazy encoding) For all $L \in \mathcal{L}^0$, $[\![L]\!]u$ is r-determinate, and one of the following conditions holds:

A. $L \Downarrow L'$ and $[\![L]\!]u \Downarrow P'$, where

$$L' \equiv \lambda y M\{\widetilde{N}/\widetilde{x}\} \qquad \text{and} \qquad P' \equiv (\widetilde{x})\Big([\![\lambda y M]\!]u \mid [\![\widetilde{x}:=\widetilde{N}]\!]\Big)$$

B. $L\Uparrow$ and $[\![L]\!]u\Uparrow$. ∎

Let us now briefly compare precongruence in the π-calculus with Abramsky's precongruence of applicative simulation for the λ-calculus.

The reduction relation $\longrightarrow$ only tells part of the story of the behaviour of a π-term P; it describes how P's parts may interact with each other, but not how P (or its parts) may interact with the environment. Without repeating all the detail from [14], we need to recall that the full behaviour of a π-term is presented by means of a labelled transition relation $\xrightarrow{\alpha}$, where α ranges over observable actions, in addition to the relation $\xrightarrow{\tau}$ (corresponding to $\rightarrow$ here), where τ is the unobservable action. Then we define $\overset{\alpha}{\Rightarrow} = \rightarrow^* \xrightarrow{\alpha} \rightarrow^*$, so that $P \overset{\alpha}{\Rightarrow}$ means 'α may be observed of P, ignoring unobservable actions'. The simplest precongruence over $\mathcal{P}$ is then defined as follows:

4.5 Definition

(1) $P \,\dot{\sqsubseteq}\, Q$ if, for all α, $P \overset{\alpha}{\Rightarrow}$ implies $Q \overset{\alpha}{\Rightarrow}$.

(2) $P \sqsubseteq Q$ if, for all contexts $\mathcal{C}[\_]$, $\mathcal{C}[P] \,\dot{\sqsubseteq}\, \mathcal{C}[Q]$. ■

This is a weak precongruence, which may be described as 'inclusion of trace-sets in all contexts'; however, for determinate π-terms – such as the translations of λ-terms – it agrees with stronger ones. But we can show that even this weak congruence is strictly stronger than $\lesssim$ as far as (the translations of) λ-terms are concerned:

4.6 Theorem (Lazy precongruence) Let $L_1, L_2 \in \mathcal{L}^0$. Then

(1) $[\![L_1]\!]u \sqsubseteq [\![L_2]\!]u$ implies $L_1 \lesssim L_2$

(2) $L_1 \lesssim L_2$ does not imply $[\![L_1]\!]u \sqsubseteq [\![L_2]\!]u$ ■

The proof of (1) is quite short, using our encoding theorem. For (2), one can adapt a counterexample of Ong [17], which strengthens Abramsky's result that his canonical model of the lazy λ-calculus is not fully abstract [1].

5 The call-by-value λ-calculus

We shall now, more briefly, repeat our programme for the *call-by-value* λ-calculus [18], where reduction in $\mathcal{L}^0$ may only occur when the argument is an abstraction. The terms $\mathcal{L}$ are as before, and it is also convenient to define the *values* $\mathcal{V}$ by

$$V \quad ::= \quad x \ \mid \ \lambda x M$$

We shall let U, V, W range over $\mathcal{V}$.

5.1 Definition The *call-by-value reduction relation* $\rightarrow_{\mathrm{v}}$ is the smallest which satisfies the rules

$$\beta_{\mathrm{v}}: \quad (\lambda x M)V \rightarrow_{\mathrm{v}} M\{^V/x\}$$

$$\text{APPL}: \quad \frac{M \rightarrow_{\mathrm{v}} M'}{MN \rightarrow_{\mathrm{v}} M'N} \qquad\qquad \text{APPR}: \quad \frac{N \rightarrow_{\mathrm{v}} N'}{MN \rightarrow_{\mathrm{v}} MN'}$$ ■

Reduction $\rightarrow_{\mathrm{v}}$ is not determinate,[3] but it is well-known that convergence $\Downarrow_{\mathrm{v}}$ is; if $M \Downarrow_{\mathrm{v}} M'$ then M' is unique. Moreover, convergence is *strong*: if $M \Downarrow_{\mathrm{v}} M'$ then all reduction sequences from M are finite. (The definitions of $\Downarrow_{\mathrm{v}}$ and $\Uparrow_{\mathrm{v}}$ are analogous with those for the lazy calculus.)

The corresponding applicative simulation relation $\lesssim_{\mathrm{v}}$ is again a precongruence[4], and $(\lambda x M)V \eqsim_{\mathrm{v}} M\{V/x\}$.

5.2 Fact $\lesssim$ and $\lesssim_{\mathrm{v}}$ are incomparable.
Proof $\mathbf{I} \lesssim \mathbf{K I \Omega} \not\lesssim \mathbf{\Omega}$, and $\mathbf{I} \not\lesssim_{\mathrm{v}} \mathbf{K I \Omega} \lesssim_{\mathrm{v}} \mathbf{\Omega}$. ■

We now turn to encoding in π-calculus. We shall continue to let x, y, z range over $\mathcal{X}$ and now let p, q, r, u, v, w range over $\mathcal{N} - \mathcal{X}$. In our new encoding $[\![M]\!]_{\mathrm{v}} p$, the name p will have a different significance. The reason is that two 'events' which coincided for the lazy calculus must now be separated, namely

- the signal at p that M has reduced to a value (needed when M is the *argument* of an application);
- the receipt of arguments by an abstraction M (needed when M is *applied*).

Further, our 'environment entries' will now contain only values. So we begin by defining $[\![y := V]\!]_{\mathrm{v}}$:

$$\begin{aligned} [\![y := \lambda x M]\!]_{\mathrm{v}} &\stackrel{\mathrm{def}}{=} !y(w).w(x)(p).[\![M]\!]_{\mathrm{v}} p \\ [\![y := x]\!]_{\mathrm{v}} &\stackrel{\mathrm{def}}{=} !y(w).\overline{x}w \end{aligned}$$

Now the first action of a (translated) value, $[\![V]\!]_{\mathrm{v}} p$, must be to announce its valuehood, thus providing access to an 'environment entry'. Note that $[\![y := V]\!]_{\mathrm{v}}$ is here a *subterm* of $[\![V]\!]_{\mathrm{v}} p$; whereas the opposite was true in the lazy encoding. And, in contrast with the lazy calculus, the translation $[\![MN]\!]_{\mathrm{v}} p$ of an application must allow M and N to 'run' in parallel:

$$\begin{aligned} [\![V]\!]_{\mathrm{v}} p &\stackrel{\mathrm{def}}{=} \overline{p}(y).[\![y := V]\!]_{\mathrm{v}} \qquad (y \text{ not free in } V) \\ [\![MN]\!]_{\mathrm{v}} p &\stackrel{\mathrm{def}}{=} (q)(r)\big(\mathbf{ap}(p,q,r) \mid [\![M]\!]_{\mathrm{v}} q \mid [\![N]\!]_{\mathrm{v}} r\big) \\ \mathbf{ap}(p,q,r) &\stackrel{\mathrm{def}}{=} q(y).\overline{y}(v).r(z).\overline{v}zp \end{aligned}$$

We now define the property which we wish $[\![M]\!]_{\mathrm{v}} p$ to possess, in place of determinacy:

5.3 Definition P is *weakly determinate* if, whenever $P \rightarrow^* Q$, then

(i) If $Q \rightarrow Q_1$ and $Q \rightarrow Q_2$, then either $Q_1 \equiv Q_2$ or $Q_1 \rightarrow Q'$ and $Q_2 \rightarrow Q'$ for some Q'.

(ii) If $Q \xrightarrow{\alpha}$ for $\alpha \neq \tau$, then $Q \not\rightarrow$. ■

We can now present a characterisation, almost exactly the same as for the lazy λ-calculus, of the way in which reduction in π-calculus matches reduction in λ-calculus:

[3]Plotkin's reduction rules [18] were more determinate than here, but the difference is unimportant.

[4]For this, we have to prove: If $M \lesssim_{\mathrm{v}} N$ then $\mathcal{C}[M]\Downarrow_{\mathrm{v}}$ implies $\mathcal{C}[N]\Downarrow_{\mathrm{v}}$. Allen Stoughton has pointed out that there is a simple direct proof of the corresponding 'context lemma' for lazy λ-calculus, following Berry and Levy [4] or Milner [12]; the same holds (with a little more trouble) for the call-by-value case.

5.4 Theorem (Call-by-value encoding) For all $L \in \mathcal{L}^0$, $[\![L]\!]_{\mathrm{v}}p$ is weakly determinate, and one of the following conditions holds:

A. $L\!\downarrow_{\mathrm{v}}\!V$ and $[\![L]\!]_{\mathrm{v}}p\!\downarrow\! P$, where

$$V \equiv W\{\tilde{U}/\tilde{x}\} \qquad \text{and} \qquad P \equiv (\tilde{x})\Big([\![W]\!]_{\mathrm{v}}p \mid [\![\tilde{x}:=\tilde{U}]\!]_{\mathrm{v}}\Big)$$

B. $L\!\uparrow_{\mathrm{v}}$ and $[\![L]\!]_{\mathrm{v}}p\!\uparrow$. ∎

We also find the same relationship of precongruences as in the lazy case:

5.5 Theorem (Call-by-value precongruence) Let $L_1, L_2 \in \mathcal{L}^0$. Then

(1) $[\![L_1]\!]_{\mathrm{v}}p \sqsubseteq [\![L_2]\!]_{\mathrm{v}}p$ implies $L_1 \lesssim_{\mathrm{v}} L_2$

(2) $L_1 \lesssim_{\mathrm{v}} L_2$ does not imply $[\![L_1]\!]_{\mathrm{v}}p \sqsubseteq [\![L_2]\!]_{\mathrm{v}}p$ ∎

The proof of (1) is just as in Theorem 4.3. For (2), consider the following:

$$\begin{aligned} L_1 &\overset{\text{def}}{=} \lambda x\Big((x\mathbf{I})(x\mathbf{K})\Big) \\ L_2 &\overset{\text{def}}{=} \lambda x\Big((\lambda y\, y(x\mathbf{K}))(x\mathbf{I})\Big) \\ L_3 &\overset{\text{def}}{=} \lambda x\Big((\lambda y\, (x\mathbf{I})y)(x\mathbf{K})\Big) \end{aligned}$$

They are all equivalent under $\eqsim_{\mathrm{v}}$. But in L_2, x will be applied to **I** first, while in L_3 it will be applied to **K** first. (In L_1 either may happen.) So in $\mathcal{P}$ we construct a fickle 'function' which behaves differently on successive calls; it will behave like **KI** the first time it is called, and like **I** the second time. When (the encodings of) L_2 and L_3 are 'applied' to the fickle function, the results will be respectively (the encodings of)

$$\begin{aligned} (\mathbf{KII})(\mathbf{IK}) &\eqsim_{\mathrm{v}} \mathbf{K} \\ (\mathbf{II})(\mathbf{KIK}) &\eqsim_{\mathrm{v}} \mathbf{I} \end{aligned}$$

In fact, we define

$$\mathbf{fickle}(r) \overset{\text{def}}{=} \bar{r}(y).y(u).\Big(u(x)(p).[\![\mathbf{KI}]\!]_{\mathrm{v}}p \mid y(v).v(x)(p).[\![\mathbf{I}]\!]_{\mathrm{v}}p\Big)$$

and place each $[\![L_i]\!]_{\mathrm{v}}q$ in the context

$$(q)(r)\Big(\mathbf{ap}(p,q,r) \mid \_ \mid \mathbf{fickle}(r)\Big)$$

6 Conclusion

We have only begun here to explore the treatment of functions in the π-calculus; the reader will already have posed many questions. For example: Exactly what is the pre-order $\sqsubseteq$ induced upon λ-terms by $L \sqsubseteq M \overset{\text{def}}{=} [\![L]\!]u \sqsubseteq [\![M]\!]u$? And are other reduction strategies easy to encode?

On the latter point, close examination of strategies reveals what may be called an oddity, seen in the light of the object paradigm. Consider any strategy in which all the rules β, APPL and APPR hold. Suppose that $M[x,x]$ is a term in which x occurs twice not within an abstraction, and suppose

$$N \equiv N_1 \to N_2 \to \cdots \to N_k \to \cdots$$

Then of course

$$(\lambda x M)N \to \cdots \to (\lambda x M)N_k \to M[N_k, N_k] \to^* M[N_{k+i}, N_{k+j}] \to \cdots$$

For the first k steps, N's reduction is 'shared'; thereafter, two separate reductions of N_k can continue within M, at different speeds (and in different directions too). This familiar situation looks odd if N is modelled as an agent; why should it clone into two or more copies just because *access* to N is transmitted through x? (Of course, it is reasonable for N to clone when it is eventually *applied* to two or more different arguments within M.) Naturally, the strategies which have been most deeply studied are those most easily expressed using the *textual substitution* which is basic to λ-calculus. One effect of providing π-calculus as a substrate may be to intensify the study of other strategies, such as those with shared reductions.

As far as application is concerned, we hope that the results of this paper will throw some light on the semantics of programming languages which contain both concurrency and non-trivial use of procedures or functions.

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] Berry, G., *Modèles Complètement Adéquats et Stables des lambda-calcul typés*, Thèse se Doctorat d'Etat, Université Paris VII, 1979.

[5] Berry, G. and Boudol, G., *The Chemical Abstract Machine*, to appear in Proc 17th Annual Symposium on Principles of Programming Languages, 1990.

[6] Boudol, G., *Towards a Lambda-Calculus for Concurrent and Communicating Systems*, Proc TAPSOFT 1989, Lecture Notes in Computer Science 351, Springer-Verlag, pp149–161, 1989.

[7] Curry, H.B. and Feys, R., **Combinatory Logic, Vol 1**, North Holland, 1958.

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

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

[10] Girard, J.-Y., *Linear Logic*, Journal of Theoretical Science, Vol 50, pp111–102, 1987.

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

[12] Milner, R., *Fully Abstract Models of Typed Lambda-calculi*, Journal of Theoretical Science, Vol 5, pp1–23, 1977.

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

[14] Milner, R., *Functions as Processes*, Internal Report, INRIA, Sophia Antipolis, December 1989.

[15] Milner, R., Parrow, J.G. and Walker, D.J., *A Calculus of Mobile Processes, Parts I and II*, Report ECS-LFCS-89-85 and -86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.

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

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

[18] Plotkin, G.D., *Call-by-name and Call-by-value and the λ-calculus*, Journal of Theoretical Science, Vol 1, pp125–159, 1975.

[19] Thomsen, B., *A Calculus of Higher-order Communicating Systems*, Proc 16th Annual Symposium on Principles of Programming Languages, pp143–154, 1989.