The π-Calculus in Direct Style

Gérard Boudol
INRIA Sophia-Antipolis, BP 93
06902 SOPHIA ANTIPOLIS CEDEX, FRANCE
email: gbo@sophia.inria.fr

Abstract

We introduce a calculus which is a direct extension of both the λ and the π calculi. We give a simple type system for it, that encompasses both Curry's type inference for the λ-calculus, and Milner's sorting for the π-calculus as particular cases of typing. We observe that the various continuation passing style transformations for λ-terms, written in our calculus, actually correspond to encodings already given by Milner and others for evaluation strategies of λ-terms into the π-calculus. Furthermore, the associated sortings correspond to well-known double negation translations on types. Finally we provide an adequate CPS transform from our calculus to the π-calculus. This shows that the latter may be regarded as an "assembly language", while our calculus seems to provide a better programming notation for higher-order concurrency.

1. Introduction

Introducing his book on "*Compiling with Continuations*" [3], Andrew Appel states that "*The beauty of* FORTRAN *– and the reason it was an improvement over assembly language – is that it relieves the programmer of the obligation to make up names for intermediate results*". We would like here to make a similar step, smaller indeed, starting from Milner's π-calculus [16,17,18], or more precisely from the very Core of PICT as our assembly language.

There has been by now a number of experiments in combining functional and concurrent features into a programming language. Typical examples are CML [22] and FACILE [12], that mix CCS and ML. Another trend is to build directly on some version of the π-calculus, exploiting the fact that it is expressive enough to encode higher-order features [17,23,24]. Examples of this are PICT [20,28] and OZ [26]. This is the approach we are interested in here. We wish to argue that the π-calculus is perhaps not the best choice as a basis for a programming language (we are not concerned with implementation issues, like the ones which led Fournet and Gonthier to develop the JOIN calculus [11]). The PICT programming language of Pierce and Turner is built upon an asynchronous, polyadic π-calculus, without sum and matching, given by the following grammar:

$$P ::= \bar{u}v_1 \cdots v_k \mid u(v_1, \ldots, v_k)P \mid !u(v_1, \ldots, v_k)P \mid (P \mid P) \mid (\nu u)P$$

As we said, this simple basic calculus, where only names are passed around, allows to encode the λ-calculus, and more generally the higher-order π-calculus of Sangiorgi [23,24]. This may seem surprising, because higher-order calculi involve the remarkably complex operation of substitution of "programs" for variables.

To introduce our work, let us see this point in some detail. A first step towards the encoding is taken by translating the λ-calculus into another one, which is both an enrichment and a restriction. The restriction is that the argument in an application must be a variable, not a compound term. The enrichment consists in adding a "where" or "let" notation, as in Landin's ISWIM. We use a different syntax, however, namely (def $x = N$ in M), to emphasize the fact that such a declaration is always recursive. Moreover, the let notation is usually associated with a call-by-value evaluation mechanism, which we do not follow. Here we deal with the weak, call-by-name λ-calculus (see [21] and [1]). That is, the reduction relation $M \to_\ell M'$ on λ-terms is given by the two rules:

$$(\lambda x M)N \to_\ell [N/x]M$$
$$M \to_\ell M' \;\Rightarrow\; MN \to_\ell M'N$$

This is a "program passing" calculus, because in a substitution $[N/x]M$, one replaces a variable by a term. The intermediate calculus – let us call it $\lambda^\star$, or the *name passing* λ-calculus – is described, rather informally, in Table 1. There are two basic reduction rules:

(i) "*small*" β-*reduction*, where a variable is instanciated with another variable, not a compound term;
(ii) *resource fetching*, where a value for a variable x is fetched whenever x occurs in the head position.

Now there is a simple translation from λ to $\lambda^\star$, given by Launchbury in [14]:

$$x^\star = x$$
$$(\lambda x M)^\star = \lambda x M^\star$$
$$(MN)^\star = (\text{def } v = N^\star \text{ in } (M^\star v)) \qquad (v \text{ fresh})$$

This work has been partly supported by FRANCE TÉLÉCOM, CTI-CNET 95-1B-182, Modélisation de Systèmes Mobiles.

POPL 97, Paris, France

$$L ::= x \mid \lambda x L \mid (Lx) \mid (\mathsf{def}\ x = L\ \mathsf{in}\ L) \qquad \textit{syntax}$$

$$x \neq z \ \Rightarrow\ (\mathsf{def}\ x = N\ \mathsf{in}\ L)z \equiv (\mathsf{def}\ x = N\ \mathsf{in}\ Lz) \qquad \textit{structural congruence}$$

$$(\lambda x L)z \to [z/x]L \qquad \textit{reduction}$$

$$(\mathsf{def}\ \cdots x = L \cdots\ \mathsf{in}\ xy_1 \cdots y_n) \to (\mathsf{def}\ \cdots x = L \cdots\ \mathsf{in}\ Ly_1 \cdots y_n)$$

$$L \to L' \ \Rightarrow\ (Lz) \to (L'z) \qquad \textit{context rules}$$

$$L \to L' \ \Rightarrow\ (\mathsf{def}\ x = N\ \mathsf{in}\ L) \to (\mathsf{def}\ x = N\ \mathsf{in}\ L')$$

$$L \to L' \ \&\ N \equiv L \ \Rightarrow\ N \to L'$$

Table 1: the Name Passing λ-Calculus

Notice that the definitions $v = L$ used in this translation are actually non recursive. This property is invariant by reduction. For terms written with non recursive definitions, it makes sense to define a "readback" mapping to λ-terms, reading $(\mathsf{def}\ v = N\ \mathsf{in}\ M)$ as $[N/v]M$. Then one can prove the following:

CORRECTNESS. *For any closed λ-term M, M has a λ_ℓ-normal form V (which is an abstraction $\lambda x N$) if and only if M^* has a λ^*-normal form W (which is a "closure"), and moreover in this case W reads back to V.*

This gives a precise meaning to the statement that the weak call-by-name λ-calculus is a "name-passing" calculus.

Now we can encode the λ-calculus into the π-calculus using λ^* as an intermediate syntax. That is, assuming that the set $\mathcal{X}$ of λ-variables is contained in the set $\mathcal{N}$ of π-names, we define a translation $[\![\cdot]\!]: \lambda^* \to (\mathcal{N} \to \pi)$ as follows:

$$[\![x]\!]u = \bar{x}u$$
$$[\![\lambda x L]\!]u = u(x,v)[\![L]\!]v$$
$$[\![Lx]\!]u = (\nu v)([\![L]\!]v \mid \bar{v}xu)$$
$$[\![\mathsf{def}\ x = N\ \mathsf{in}\ L]\!]u = (\nu x)([\![L]\!]u \mid !x(v)[\![N]\!]v)$$

This is a slight variation of Milner's encoding [17,18]. The interesting point to note here is that there is an *exact, step by step* correspondence between reductions of L in λ^* and reductions of $[\![L]\!]u$ in π.

The trailing name u in the translation $[\![M^*]\!]u$ is somewhat irritating. It prevents a *direct* representation of data, especially of higher-order kind: there is nothing like a "function" in the π-calculus, simply because everything must be "located" at some input (or output) channel. Put in another way, there is, as far as I know, no notion of type for π-processes. The best we can say is that a process is *ok* with respect to a *sorting*, which assigns types to channel names, not to agents. As a matter of fact, a large part of the development of the π-calculus as a programming language ([20]; see also [11]) consists in introducing derived forms that get round the inconveniences of a "continuation passing style", and of the ubiquitous "result channel".

Our aim here is to design a *direct* model for higher-order, untyped concurrency, that would have the same expressive power as the π-calculus, but would be more convenient as a programming notation. The idea is very simple: we start from λ^*, which is quite close to both λ – as far as the syntax is concerned – and π – regarding the reduction relation –, and we relax its syntax to make it closer to the π-calculus. The calculus we will get, which we call *the blue calculus*, or π^*, contains both the λ and the (sorted) π calculi in a very direct and simple way. These direct embeddings deal with simple types and simple sorts as well: we design a natural simple type system for our calculus that extends both Curry's type system for λ and Milner's (simple) sorting system for π. Furthermore, we show that, assuming a simple discipline on the use of names, there is a *continuation passing style transform* within our calculus, whose target is actually the π-calculus, justifying our claim that our calculus is "the π-calculus in direct style" – or conversely, that π is a continuation passing calculus.

Let us briefly explain how we build the blue calculus out of λ^*, using the π-calculus as a guideline. The first step is to break a term $(\mathsf{def}\ x = N\ \mathsf{in}\ L)$ into two parts; one is the *declaration* itself, which we write $\langle x = N\rangle$, or in a more concrete syntax $(\mathsf{decl}\ x = N)$, and the other one is the *scope* of the name x, which is determined by a (νx) guard. Following the chemical metaphor [4], this means that declarations are now "molecules" which float in the same "soup" as the main agent L, that is in parallel with it. Namely, $(\mathsf{def}\ x = N\ \mathsf{in}\ L)$ is now written $(\nu x)(\langle x = N\rangle \mid L)$. Incidentally, we have introduced parallel composition as a construct of our model – this should not be too surprising.

The only remaining ingredient is that of *single resources*. In [6,8] we devised a *λ-calculus with resources* that was intended to model by λ-calculus means the discriminating power of the π-calculus. There we found it useful to consider a variant of $\langle x = P\rangle$, that we denote $\langle x \Leftarrow P\rangle$ or $(\mathsf{decl}\ x \Leftarrow P)$, which means that P may be used *only once* as a value for x – by contrast, P is an "inexhaustible resource" for x in $\langle x = P\rangle$. The construct $\langle x \Leftarrow P\rangle$, together with branching features that we briefly discuss in the conclusion, is useful to model processes with a mutable state. To sum up, the syntax of π^* is:

$$P ::= x \mid (\lambda x)P \mid (Px) \mid \langle x = P\rangle \mid \langle x \Leftarrow P\rangle \mid (P \mid P) \mid (\nu x)P$$

We write $(\lambda x)P$, as Milner does in [18], to recall that this is a "small" abstraction, where x stands for variables. Besides the "small β-rule", there is a "resource fetching" rule, whose effect is

$$(xz_1 \cdots z_k \mid \langle x \Leftarrow R\rangle) \to Rz_1 \cdots z_k$$

Then we may describe our computational model as follows:

> *besides local β-reduction that may produce values of the form $(\lambda x)P$, computing consists in sending asynchronous messages $xz_1 \cdots z_k$ that call for "resources" or "services" R for x, which are found in the environment in the form of declarations $\langle x \Leftarrow R\rangle$. This results in the application $Rz_1 \cdots z_k$ of the service to the arguments of the message.*

This sounds perhaps more familiar than "computing by channel passing", though everything written following this latter style, or more precisely everything written in the π-calculus, and in the λ-calculus as well, can be read directly within our model.

To conclude this introduction, let us see an example, illustrating the difference with the "channel passing style" of the π-calculus. Milner in [18] introduced a representation of *lists* of data, whose constructors may be declared, in the blue calculus, in a global environment as follows:

$$\langle \mathsf{nil} = (\lambda nc)n\rangle$$
$$\langle \mathsf{cons} = (\lambda ht)(\lambda nc)cht\rangle$$

Here it would be useful to have some syntactic sugar, writing $\langle \mathsf{cons}(h,t) = (\lambda nc)cht\rangle$ for instance. Then this encoding of lists may be understood as follows: a list is a function taking as argument a pair, made of a "nil cell" n and a "cons cell" c, and returning n in the case of nil, and the "cons" of the head h and the tail t otherwise. One should notice that the usual "if-then-else" combinator of the λ-calculus, that is

$$\langle \mathsf{cond} = (\lambda bxy)\, bxy\rangle$$

is such that, garbage collecting inaccessible declarations:

$$\mathsf{cond}\,\ell xy \qquad \text{reduces to} \quad \begin{cases} x & \text{if } \ell = \mathsf{nil} \\ yht & \text{if } \ell = \mathsf{cons}\,ht \end{cases}$$

Then the append function on lists may be defined as follows:

$$\begin{aligned}\langle \mathsf{append} = (\lambda xy)(\nu v)(\mathsf{cond}\,xyv \mid \\ \langle v \Leftarrow (\lambda ht)(\nu r)(\mathsf{cons}\,hr \mid \\ \langle r \Leftarrow \mathsf{append}\,ty\rangle)))\rangle\end{aligned}$$

For the purpose of comparison with the π-calculus encoding [18], we have compelled ourselves to write this in the core syntax of our calculus, without using any syntactic sugar. We think that even if this is a little verbose, our definition of append looks like the one we are used to. Obviously, using programming language notations like the "where" or "def in" constructs would ease reading the term. We could also introduce, as Milner does in [18], a case construct

$$\begin{aligned}&\mathsf{case}\ \ell\ \mathsf{of} : \mathsf{nil} \Rightarrow P \\ &\qquad : \mathsf{cons}(h,t) \Rightarrow Q\end{aligned}$$

as a notation for

$$\begin{aligned}(\nu pq)\big(\mathsf{cond}\,\ell pq \mid &\langle p \Leftarrow P\rangle \mid \\ &\langle q \Leftarrow (\lambda ht)Q\rangle\big)\end{aligned}$$

Alternatively, we may also write, using the application construct (PQ) of the λ-calculus as an abbreviation for $(\nu z)(Pz \mid \langle z = Q\rangle)$:

$$\langle \mathsf{append} = (\lambda xy)\mathsf{cond}\,xy\,((\lambda ht)\mathsf{cons}\,h(\mathsf{append}\,ty))\rangle$$

though we lose the explicit indication that the compound arguments are to be used at most once. One would define similarly a map function, taking a function f and a list ℓ as arguments, and computing the list of applications of f to the items of ℓ:

$$\langle \mathsf{map} = (\lambda f\ell)\mathsf{cond}\,\ell\ell\,((\lambda ht)\mathsf{cons}\,(fh)(\mathsf{map}\,ft))\rangle$$

This shows that our calculus supports the usual style of functional programming. Moreover, we do not have to resort to type encodings to provide nil, cons, append, and so on, with the type they usually have.

2. The Computational Model

We assume given a denumerable set $\mathcal{N}$ of names, ranged over by $u, v, w\ldots$ We sometimes use also $x, y, z\ldots$ when we have in mind variables rather than names, though there is no formal distinction here. For notational convenience, we distinguish three syntactic categories in the grammar of our calculus:

$$\begin{array}{lll} P ::= A \mid D \mid (P \mid P) \mid (\nu u)P & & \textit{processes} \\ A ::= u \mid (\lambda u)P \mid (Pu) & & \textit{agents} \\ D ::= \langle u \Leftarrow P\rangle \mid \langle u = P\rangle & & \textit{declarations} \end{array}$$

where $u \in \mathcal{N}$ is any name. We let $P, Q, R\ldots$ range over processes. We use the standard abbreviations from the λ-calculus, namely $Pv_1 \cdots v_k$ for $(\cdots(Pv_1)\cdots v_k)$ and $(\lambda u_1 \ldots u_k)P$ for $(\lambda u_1)\ldots(\lambda u_k)P$, and similarly $(\nu u_1 \ldots u_k)P$ for $(\nu u_1)\ldots(\nu u_k)P$. Sometimes we add some parentheses, as in $((\lambda u)P)v$ or in $((\nu u)P)v$, sometimes we omit the brackets $\langle\cdot\rangle$, writing simply $u \Leftarrow P$ and $u = P$, to ease reading the terms. As usual, (λu) binds u, and similarly u is bound in $(\nu u)P$, whereas it is free in $\langle u \Leftarrow P\rangle$ and $\langle u = P\rangle$. We denote by $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$ respectively the sets of *free* and *bound* names of P, and by $[v/u]P$ the result of substituting the name v for u in P. This may involve α-conversion, that is renaming bound names of P, to avoid capturing v. We denote the congruence generated by this kind of renaming by $P =_\alpha Q$.

A *π^*-context* is a term written using the same syntax as for π^*-terms, plus a constant $[]$, the hole. We use boldface capital letters $\mathbf{A}, \mathbf{B}, \mathbf{C}\ldots$ to denote contexts. Filling the hole in $\mathbf{C}$ with a π^*-term P results in a π^*-term denoted $\mathbf{C}[P]$. Notice that some free names of P may be bound by the context in $\mathbf{C}[P]$. The notion of substitution is extended to contexts in the obvious way, namely $[v/u][] = []$. We shall use a particular kind of contexts, which we call *evaluation contexts*, where the hole does not occur within an abstraction or a declaration. That is, the set $\mathcal{E}$ of these contexts is given by the following grammar:

$$\mathbf{E} ::= [] \mid (\mathbf{E}u) \mid (\mathbf{E} \mid P) \mid (P \mid \mathbf{E}) \mid (\nu u)\mathbf{E}$$

structural equivalence:

$$(P \mid Q) \equiv (Q \mid P) \qquad \textit{commutativity}$$
$$((P \mid Q) \mid R) \equiv (P \mid (Q \mid R)) \qquad \textit{associativity}$$
$$((\nu u)P \mid Q) \equiv (\nu u)(P \mid Q) \quad (u \text{ not free in } Q) \qquad \textit{scope migration}$$
$$(P \mid Q)u \equiv (Pu \mid Qu) \qquad \textit{distributivity}$$
$$((\nu u)P)v \equiv (\nu u)(Pv) \quad (u \neq v)$$
$$Du \equiv D$$
$$\langle u = P\rangle \equiv \langle u \Leftarrow (P \mid \langle u = P\rangle)\rangle \qquad \textit{duplication}$$
$$P \equiv Q \;\Rightarrow\; \mathbf{E}[P] \equiv \mathbf{E}[Q]$$

reduction:

$$((\lambda u)P)v \to [v/u]P \qquad (\beta)$$
$$(u \mid \langle u \Leftarrow P\rangle) \to P \qquad \varrho \quad \textit{(resource fetching)}$$
$$P \to P' \;\Rightarrow\; \mathbf{E}[P] \to \mathbf{E}[P'] \qquad \textit{context rule}$$
$$P \to P' \;\&\; Q \equiv P \;\Rightarrow\; Q \to P'$$

Table 2: Structure and Reduction

$$(\nu u_1 \ldots u_n)(\underbrace{V_1 \mid \cdots \mid V_m}_{\text{abstractions}} \mid \underbrace{M_1 \mid \cdots \mid M_s}_{\text{messages}} \mid \underbrace{\langle v_1 \Leftarrow R_1\rangle \mid \cdots \mid \langle v_r \Leftarrow R_r\rangle}_{\text{named resources}})$$

Figure: Canonical Form

where P is any term. We use $\mathbf{E}$, $\mathbf{F}\ldots$ to range over $\mathcal{E}$.

Regarding the operational semantics, we follow a CHAM style [4]. That is, we define a *structural equivalence* $P \equiv Q$, allowing us to regard each process as a "soup" or a "chemical solution", and we define the reduction relation with this intutitive representation in mind. The operational description of the calculus is written in Table 2. We have omitted from the table the rule stating that $\equiv$ is an equivalence, and that it contains α-conversion. Together with the rule that structural equivalence is compatible with evaluation contexts, the first three rules allow us to write, for any term P

$$P \;\equiv\; (\nu u_1 \ldots u_k)(A_1 \mid \cdots \mid A_n \mid D_1 \mid \cdots \mid D_r)$$

That is, any process is represented as a "chemical solution" where the floating molecules are either agents A_i or declarations D_j, and the scope of (top level) private names is global. Then there are three further structural manipulations on agents, for pushing application to a name (Qu) through the structure of Q, so that any term may be turned into an equivalent one where the agents A_i have a simple form, namely applications of abstractions or names to a series of arguments, i.e. $A_i = ((\lambda u)P)v_1 \cdots v_m$ or $A_i = uv_1 \cdots v_m$, what we call a *message*. Accordingly, two kinds of reduction may occur, called (β) and ϱ in the Table 2. The *small β-reduction* $P \to_{(\beta)} Q$, that is the reduction without resource fetching, enjoys the following properties:

LEMMA 2.1.

(i) *the (β)-reduction satisfies the diamond property: if $P \to_{(\beta)} P_0$ and $P \to_{(\beta)} P_1$ then either $P_0 \equiv P_1$ or $P_0 \to_{(\beta)} P'$ and $P_1 \to_{(\beta)} P'$ for some P'*

(ii) *the (β)-reduction is (strongly) normalizing: for any P there exists Q such that $P \stackrel{*}{\to}_{(\beta)} Q$ and Q is (β)-irreducible, that is $\{Q' \mid Q \to_{(\beta)} Q'\} = \emptyset$.*

For any term P we denote by $\mathrm{nf}_\beta(P)$ the (β)-normal form of P. Now, combining structural transformations together with (β)-normalization, we get from any term P another one in *canonical form*, shown in the figure above.

We denote by $P \to_\varrho Q$ the reduction relation generated by the resource fetching rule ϱ. For instance, a message sent to an inexhaustible resource is processed as follows:

$$\begin{aligned}(vw_1 \cdots w_p \mid \langle v = R\rangle) &\equiv (v \mid v \Leftarrow (R \mid v = R))w_1 \cdots w_p \\ &\to_\varrho (R \mid \langle v = R\rangle)w_1 \cdots w_p \\ &\equiv (Rw_1 \cdots w_p \mid \langle v = R\rangle)\end{aligned}$$

Admittedly, using distributivity from right to left is just a trick, and we could better assume that the resource fetching rule is, as in $\lambda^\star$:

$$(uv_1 \cdots v_n \mid \langle u \Leftarrow P\rangle) \to Pv_1 \cdots v_n$$

Indeed, we could then distinguish "heating" and "cooling" structural manipulations, as in [4], since we mainly use the rules (except commutativity and associativity) from left to right, "heating" a process to prepare a reduction. We can assume that (β)-normalization is performed as a preliminary phase of evaluation, since we have:

LEMMA 2.2. *Resource fetching and (β)-reduction commute: if $P \to_\varrho P_0$ and $P \to_{(\beta)} P_1$ then $P_0 \to_{(\beta)} P'$ and $P_1 \to_\varrho P'$ for some P'.*

Clearly the resource fetching part of the reduction is responsible for the expressive power – this is perhaps why (β)-reduction was neglected in the work of Milner [18] and Sangiorgi [23,24]. For instance resource fetching *may not terminate*. The simplest example is given by the term

$$\Omega =_{\text{def}} (\nu u)(u \mid \langle u = u \rangle)$$

Resource fetching, together with the commutativity of parallel composition, is also responsible for introducing "critical pairs", or *conflicts*, like in

$$(P \oplus Q) =_{\text{def}} (\nu u)(u \mid \langle u = P \rangle \mid \langle u = Q \rangle) \quad \text{where} \quad u \notin \mathsf{fn}(P) \cup \mathsf{fn}(Q)$$

or $(uv_1 \cdots v_k \mid uw_1 \cdots w_n \mid \langle u \Leftarrow R \rangle)$. The first form of non-determinism should be avoided (if a "service" does not work properly, one should be able to locate the defective resource), while the second one is inherent in asynchronous distributed systems.

$(\lambda^* \subset \pi^*)$ We announced in the introduction that our calculus contains the λ-calculus in a direct way. This is very easy to see: firstly, the λ^*-terms may be regarded as elements of a subset of π^*, given by the grammar

$$L ::= x \mid (\lambda x)L \mid (Lx) \mid (\nu x)(L \mid \langle x = L \rangle)$$

The structural law of λ^* is still valid in π^* – it is now written $((\nu u)(P \mid D))v \equiv (\nu u)(Pv \mid D)$ if $u \neq v$ –, and reducing a λ^*-term is the same in both calculi, up to structural equivalence. Then we can translate the λ-calculus into the blue calculus in the obvious way:

$$\begin{aligned} [\![x]\!] &= x \\ [\![\lambda x M]\!] &= (\lambda x)[\![M]\!] \\ [\![MN]\!] &= (\nu v)([\![M]\!]v \mid \langle v = [\![N]\!] \rangle) \qquad (v \text{ fresh}) \end{aligned}$$

The translation may be slightly optimized, if we define $[\![MN]\!]$ to be $[\![M]\!]x$ whenever N is the variable x. Now we can regard the λ-calculus as a sub-calculus of π^*, using the notation MN. More generally, we will use (PQ) as an abbreviation for $(\nu v)(Pv \mid \langle v = Q \rangle)$, provided that v does not occur in P or Q. We may also use the standard notations for the usual combinators, like for instance $\mathbf{T} = \mathbf{K} = (\lambda xy)x$ and $\mathbf{F} = (\lambda xy)y$ for the truth values.

In the blue calculus one can encode combinators that are not λ-definable, like the non-determinitic choice operator, which may be written $\oplus = (\lambda xy)((\nu u)(u \mid \langle u = x \rangle \mid \langle u = y \rangle)$. Another example is the *parallel or* combinator. This is a function por that takes two arguments and returns "true" as soon as one of them is true – and both of them are "boolean" in the sense that they may either diverge or evaluate to a truth value. In the blue calculus this may be defined as follows:

$$\mathsf{por} =_{\text{def}} (\lambda xy)(\nu tf)(xtf \mid ytf \mid \langle t \Leftarrow \mathbf{T} \rangle \mid \langle f \Leftarrow \langle f \Leftarrow \mathbf{F} \rangle))$$

That this term fulfils its specification depends on the semantics we have for the calculus. This is the topic of the next section. One may notice the use of nested declarations $u \Leftarrow u \Leftarrow \cdots$ as a synchronisation mechanism in the definition of por. One can also use single resources to encode processes with a changing state. For instance, denoting $(\nu x)(x \mid x = P)$ by $\mathsf{rec}\, x.P$, one can define a "one-slot buffer" that performs alternatively "put" and "get" operations, as follows:

$$\mathsf{buff} =_{\text{def}} \mathsf{rec}\, b.\langle \mathsf{put} \Leftarrow (\lambda x)(\mathsf{get} \Leftarrow (x \mid b)) \rangle$$

The reader is invited to find out how $(\mathsf{buff} \mid \mathsf{put}\, f \mid \mathsf{get}\, v_1 \cdots v_k)$ evaluates (see the conclusion for another, similar example).

$(\pi \subset \pi^*)$ To conclude with the examples, we indicate in which sense the π-calculus may also be regarded as a sub-calculus of π^* – the technical details will be treated of in a next section. As a matter of fact, the containment of π into π^* is even more direct than for λ. The idea is that a message $\bar{u}v_1 \cdots v_n$ sent on a channel u is just the application $uv_1 \cdots v_n$ of the name u to the sequence of arguments, and that an input $u(v_1, \ldots, v_k)P$ on the communication channel u is a declaration $\langle u \Leftarrow (\lambda v_1 \ldots v_k)P \rangle$ – the reader may guess what is a replicated input. Then the π-terms are the elements of the subset of π^* given by the grammar

$$\begin{aligned} P &::= M \mid E \mid (P \mid P) \mid (\nu u)P \\ M &::= u \mid (Mu) \\ E &::= \langle u \Leftarrow F \rangle \mid \langle u = F \rangle \\ F &::= P \mid (\lambda u)F \end{aligned}$$

The π-communication is achieved as a resource fetching act followed by a sequence of (β)-reductions. However, this only works properly – that is: as in the π-calculus – for well-sorted processes, as we shall see. Nevertheless, we regard the π-calculus as a sub-calculus of π^*, and we may import most of the derived forms of PICT with only minor syntactic changes.

3. Observational Semantics

Regarding the semantics of our calculus, we adopt the standard approach, namely Morris' extensional preorder, also called *may testing*. This is a preorder $P \sqsubseteq Q$, meaning that any test that P passes successfully is also passed by Q. A *test* is a context with exactly one occurrence of the hole. We denote by $\mathcal{T}$ the set of tests – we still use $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$... to denote them. The test $\mathbf{C}$ succeeds on P if $\mathbf{C}[P]$ has a value. Usually, a value is a closed normal form with respect to some reduction strategy. For instance, in the lazy λ-calculus [1], a value is a (closed) abstraction. Here, we may have several agents computing in parallel, and therefore we regard as a value a term where at least one component is an abstraction, as in [7]. That is, a *value* is a (closed) term given by the following grammar:

$$V ::= (\lambda u)P \mid (V \mid P) \mid (P \mid V) \mid (\nu u)V$$

where P is any term. The set of values is denoted $\mathcal{V}$. It is easy to see that this set is closed under structural manipulations and reduction. We say that P *converges*, in notation $P{\Downarrow}$, if P has a value, that is $P \stackrel{*}{\rightarrow} V$ for some $V \in \mathcal{V}$. The semantic preorder is now defined as follows:

$$P \sqsubseteq Q \;\Leftrightarrow_{\text{def}}\; \forall \mathbf{C} \in \mathcal{T}.\ \mathbf{C}[P]{\Downarrow} \;\Rightarrow\; \mathbf{C}[Q]{\Downarrow}$$

This is obviously a precongruence. We denote by $\simeq$ the associated equivalence, that is

$$P \simeq Q \Leftrightarrow P \sqsubseteq Q \;\&\; Q \sqsubseteq P$$

One may notice that we would get the same semantics by allowing "free messages" to be values too. This is because, for any finite set U of names and any P such that $\mathsf{fn}(P) \subseteq U$, P "converges" in the extended sense if and only if $\mathbf{K}_U[P]\Downarrow$ where

$$\mathbf{K}_U = (\nu u_1 \ldots u_k)([] \mid \langle u_1 = \Xi\rangle \mid \cdots \mid \langle u_k = \Xi\rangle)$$

with $\Xi = (\nu r)(r \mid \langle r = (\lambda x)r\rangle)$ and $\{u_1, \ldots, u_k\} = U$.

Our aim now is to show that the observational semantics may be determined using only some restricted kind of tests, namely, that it is enough to use *evaluation contexts* to test the processes. As a matter of fact, we also need to instantiate the free names of the tested terms, therefore, using σ to denote any substitution, we define the corresponding testing semantics $\sqsubseteq^{\mathcal{E}}$ as follows:

$$P \sqsubseteq^{\mathcal{E}} Q \;\Leftrightarrow_{\mathrm{def}}\; \forall \mathbf{E} \in \mathcal{E}\; \forall \sigma.\; \mathbf{E}[\sigma P]\Downarrow \;\Rightarrow\; \mathbf{E}[\sigma Q]\Downarrow$$

It is not difficult to check that

$$P \to P' \;\Rightarrow\; \sigma P \to \sigma P'$$

and therefore one can immediately see that reduction is decreasing with respect to the " evaluation testing" preorder:

REMARK 3.1. $P \to P' \;\Rightarrow\; P' \sqsubseteq^{\mathcal{E}} P$

Moreover, the semantics is preserved by (β)-conversion:

LEMMA 3.2. $[u/x]P \sqsubseteq^{\mathcal{E}} ((\lambda x)P)u$ *and* $((\lambda x)P)u \sqsubseteq^{\mathcal{E}} [u/x]P$

The first ordering is given by the previous remark, and for the converse one uses the Lemmas 2.1 and 2.2. A consequence is that we could also define $\sqsubseteq^{\mathcal{E}}$ exactly as the observational semantics $\sqsubseteq$, but using tests of the form

$$\mathbf{E}[((\lambda x_1 \ldots x_n)[])u_1 \cdots u_n]$$

To show that $\sqsubseteq$ and $\sqsubseteq^{\mathcal{E}}$ are actually the same, we first notice the obvious fact that the latter is preserved by name substitution.

REMARK. $P \sqsubseteq^{\mathcal{E}} Q \;\Rightarrow\; \sigma P \sqsubseteq^{\mathcal{E}} \sigma Q$

LEMMA (the CONTEXT LEMMA) 3.3. $P \sqsubseteq^{\mathcal{E}} Q \;\Leftrightarrow\; P \sqsubseteq Q$

PROOF: the implication "$\Leftarrow$" is obvious. To establish the converse "$\Rightarrow$", we define the *depth* of a test $\mathbf{C} \in \mathcal{T}$, denoted $h(\mathbf{C})$, as the number of abstractions (λu) and declarations $\langle u \Leftarrow\rangle$ or $\langle u = \rangle$ that one goes through to reach the hole. In particular, $h(\mathbf{C}) = 0$ if and only if $\mathbf{C} \in \mathcal{E}$. Clearly, the depth is preserved by structural manipulations (if we disallow the use of the duplication rule $\langle u = R\rangle \equiv \langle u \Leftarrow (R \mid \langle u = R\rangle)\rangle$), to avoid duplicating the hole). We show that if $P \sqsubseteq^{\mathcal{E}} Q$ and $\mathbf{C}[P]\Downarrow$ then $\mathbf{C}[Q]\Downarrow$ by induction on $(h(\mathbf{C}), l)$, w.r.t. the lexicographic ordering, where l is the length of a reduction sequence from $\mathbf{C}[P]$ to a value. We first observe that any test $\mathbf{C}$ may be transformed using structural manipulations (without using the duplication rule) into another one having a canonical form

$$\overline{\mathbf{C}} = (\nu u_1 \ldots u_k)(\mathbf{A}_1 \mid \cdots \mid \mathbf{A}_n \mid \mathbf{D}_1 \mid \cdots \mid \mathbf{D}_r) \qquad (*)$$

where the $\mathbf{A}_i$'s and the $\mathbf{D}_i$'s are respectively given by the grammars:

$$\mathbf{A} ::= [] \mid u \mid (\lambda u)\mathbf{B} \mid (\mathbf{A}u)$$
$$\mathbf{D} ::= \langle u \Leftarrow \mathbf{B}\rangle \mid \langle u = \mathbf{B}\rangle$$

where $\mathbf{B}$ is any context. Obviously, only one of the $\mathbf{A}_i$'s and the $\mathbf{D}_i$'s contains the hole, and the others are terms, or declarations. Moreover, each $\mathbf{A}_i$ has the form $\mathbf{A}'_i v_1 \cdots v_p$ where $\mathbf{A}'_i$ is either the hole $[]$, or a name u or an abstraction context $(\lambda u)\mathbf{B}$.

However, $\mathbf{C}[P]$ is not in general structurally equivalent to $\overline{\mathbf{C}}[P]$, because in transforming $\mathbf{C}[P]$ one may have to use α-conversion, that is $(\nu u)R \equiv (\nu v)[v/u]R$ with v not in R, to perform some scope migration for instance. Nevertheless, we have: given $\mathbf{C}$ and a finite set U of names, there exists $\overline{\mathbf{C}}$ as above $(*)$, with $h(\overline{\mathbf{C}}) = h(\mathbf{C})$, and a substitution σ such that

$$\mathsf{nm}(R) \subseteq U \;\Rightarrow\; \mathbf{C}[R] \equiv \overline{\mathbf{C}}[\sigma R]$$

Since for any R such that $\mathsf{nm}(R) \subseteq U$ (and we may take U large enough, so that P and Q fulfil the condition), the terms $\mathbf{C}[R]$ and $\overline{\mathbf{C}}[\sigma R]$ have the same reduction sequences, using the previous remark we may assume for the proof that the tests under consideration have the form $(*)$.

(1) $h(\overline{\mathbf{C}}) = 0$. This case is trivial since it means $\overline{\mathbf{C}} \in \mathcal{E}$.

(2) $h(\overline{\mathbf{C}}) > 0$. Note that in this case, no $\mathbf{A}_i$ may be $[]v_1 \cdots v_p$.

(2.1) $l = 0$. There must be some i such that $\mathbf{A}_i[\sigma P]$ is an abstraction. Then either $\mathbf{A}_i$ does not contain the hole, or $\mathbf{A}_i = (\lambda u)\mathbf{C}'$. In both cases, $\mathbf{C}[Q]$ is a value too.

(2.2) $l > 0$. There exists R such that $\overline{\mathbf{C}}[\sigma P] \to R$ and R converges in $l - 1$ steps. We examine the possible cases:

(2.2.1) $\mathbf{A}_i[\sigma P] \to_{(\beta)} P'$ for some i and some P' such that

$$R \equiv (\nu u_1 \ldots u_k)(\mathbf{A}_1[\sigma P] \mid \cdots \mid P' \mid \cdots \mid \mathbf{A}_n[\sigma P] \mid \mathbf{D}_1[\sigma P] \mid \cdots \mid \mathbf{D}_r[\sigma P])$$

(a) If $\mathbf{A}_i$ does not contain the hole, then we let

$$\mathbf{C}' \equiv (\nu u_1 \ldots u_k)(\mathbf{A}_1 \mid \cdots \mid P' \mid \cdots \mid \mathbf{A}_n \mid \mathbf{D}_1 \mid \cdots \mid \mathbf{D}_r)$$

and we use the induction hypothesis on the length since $h(\mathbf{C}') = h(\mathbf{C})$ and $\mathbf{C}[S] \to \mathbf{C}'[S]$ for any S.

(b) If $\mathbf{A}_i$ is a test, we have $\mathbf{A}_i = ((\lambda u)\mathbf{B})\, v_1 \cdots v_p$ and $P' = ([v_1/u](\mathbf{B}[\sigma P]))v_2 \cdots v_p$. Then it is easy to see that there exists $\mathbf{B}'$ and a substitution σ' such that $[v_1/u](\mathbf{B}[S]) \equiv \mathbf{B}'[\sigma' S]$ for any S, therefore if we let

$$\mathbf{C}' \equiv (\nu u_1 \ldots u_k)(\mathbf{A}_1 \mid \cdots \mid \mathbf{B}' v_2 \cdots v_p \mid \cdots \mid \mathbf{A}_n \mid \mathbf{D}_1 \mid \cdots \mid \mathbf{D}_r)$$

then we may use the induction hypothesis since $h(\mathbf{C}') < h(\mathbf{C})$ and $\mathbf{C}[S] \to \equiv \mathbf{C}'[\sigma' S]$ for any S.

(2.2.2) $\mathbf{A}_i = u v_1 \cdots v_p$ and $\mathbf{D}_j = \langle u \Leftarrow \mathbf{B}\rangle$ or $\mathbf{D}_j = \langle u = \mathbf{B}\rangle$ and the first reduction step consists in fetching $\mathbf{B}[\sigma P]$ for u in $\mathbf{A}_i$. The case where $\mathbf{B}$ does not contain the hole is similar to (a) above: the reduction is independent from what we put in the hole. Otherwise, the case of $\mathbf{D}_j = \langle u \Leftarrow \mathbf{B}\rangle$ is similar to the case (b) above. So let us assume that $\mathbf{D}_j = \langle u = \mathbf{B}\rangle$,

$$\frac{}{\Gamma \vdash u : \tau}\ \Gamma(u) = \tau \qquad \frac{u:\sigma, \Gamma \vdash P : \tau}{\Gamma \vdash (\lambda u)P : \sigma \to \tau} \qquad \frac{\Gamma \vdash P : \sigma \to \tau\ ,\ \Gamma \vdash u : \sigma}{\Gamma \vdash (Pu) : \tau}$$

$$\frac{\Gamma \vdash P : \tau\ ,\ \Gamma \vdash Q : \tau}{\Gamma \vdash (P \mid Q) : \tau} \qquad \frac{\Gamma \vdash P : \tau}{\Gamma_{\lfloor u} \vdash (\nu u)P : \tau}$$

$$\frac{\Gamma \vdash Q : \sigma}{\Gamma \vdash \langle u \Leftarrow Q \rangle : \tau}\ \Gamma(u) = \sigma \qquad \frac{\Gamma \vdash Q : \sigma}{\Gamma \vdash \langle u = Q \rangle : \tau}\ \Gamma(u) = \sigma$$

Table 3: the Simple Type System

where $\mathbf{B}$ contains the hole. Then

$$R \equiv (\nu u_1 \dots u_k)(\mathbf{A}_1 \mid \cdots \mid (\mathbf{B}[\sigma P])v_1 \cdots v_p \mid \cdots \mid \mathbf{A}_n \mid \mathbf{D}_1 \mid \cdots \mid \mathbf{D}_j[\sigma P] \mid \cdots \mid \mathbf{D}_r)$$

Let

$$\mathbf{C}' \equiv (\nu u_1 \dots u_k)(\mathbf{A}_1 \mid \cdots \mid (\mathbf{B}[\sigma P])v_1 \cdots v_p \mid \cdots \mid \mathbf{A}_n \mid \mathbf{D}_1 \mid \cdots \mid \mathbf{D}_r)$$

where the hole is in $\mathbf{D}_j$. Since $\mathbf{C}'[\sigma P] \equiv R$ we have $\mathbf{C}'[\sigma Q]\Downarrow$ by induction on the length. Now let

$$\mathbf{C}'' \equiv (\nu u_1 \dots u_k)(\mathbf{A}_1 \mid \cdots \mid \mathbf{B}v_1 \cdots v_p \mid \cdots \mid \mathbf{A}_n \mid \mathbf{D}_1 \mid \cdots \mid \mathbf{D}_j[\sigma Q] \mid \cdots \mid \mathbf{D}_r)$$

(the hole is now in $\mathbf{B}$). Clearly $h(\mathbf{C}'') = h(\overline{\mathbf{C}})-1$, and $\mathbf{C}''[\sigma P] = \mathbf{C}'[\sigma Q]$, hence $\mathbf{C}''[\sigma P]\Downarrow$. Therefore using the induction hypothesis on the depth we conclude $\mathbf{C}''[\sigma Q]\Downarrow$, hence also $\mathbf{C}[Q]\Downarrow$ since $\mathbf{C}[Q] \to \equiv \mathbf{C}''[\sigma Q]$ ❑

One may observe that an evaluation test $\mathbf{E} \in \mathcal{E}$, applied to a term P, can be transformed as follows:

$$\mathbf{E}[P] \equiv (\nu u_1 \dots u_k)((\sigma P)v_1 \cdots v_p \mid A_1 \mid \cdots \mid A_n \mid D_1 \mid \cdots \mid D_r)$$

One may wonder whether one could further restrict the kind of tests that are really needed to determine the semantics. Actually, one cannot go very far on this way. One may assume for instance that the A_i's are messages: if A_j is an abstraction, then the test is useless, and if A_j contains a (β)-redex then this may be reduced locally. But messages in the testing environment of a process are nedeed to test the declarations. Without these messages, we would not be able to detect a difference between $\langle u \Leftarrow \Omega \rangle$ and $\langle u \Leftarrow (\lambda v)v \rangle$ for instance. Regarding the declarations D_j's, we could assume that they do not involve inexhaustible resources $\langle u = R \rangle$, because only a finite amount of resources is needed for the convergence of an evaluation.

The Context Lemma is very useful to prove some semantical equalities or inequalities. We leave the following to the reader:

EXERCISE 3.4.

(i) $u \notin \mathsf{fn}(P) \ \Rightarrow\ (\nu u)P \simeq P \ \ \&\ \ (\nu u)(P \mid \langle u \Leftarrow R \rangle) \simeq P$

(ii) $(\lambda u)(P \mid Q) \simeq ((\lambda u)P \mid (\lambda u)Q)$

(iii) $u \neq v \ \Rightarrow\ (\lambda u)(\nu v)P \simeq (\nu v)(\lambda u)P$

One may also easily show that η-expansion is valid, that is $P \sqsubseteq (\lambda u)(Pu)$ if $u \notin \mathsf{fn}(P)$. We shall not investigate in this paper the question of what is the semantics induced on source calculi by translations, as it is done in [25,8] for instance. Nevertheless, regarding the semantics of the λ-calculus within $\pi^\star$, we believe that the result of [9] still holds, for it is easy to encode the λ-calculi with "multiplicities" and with "resources" of [9,8] in $\pi^\star$. For instance, a "bag" of resources $R_1, \dots, R_n$ for x is represented by

$$(x \Leftarrow R_1 \mid \cdots \mid x \Leftarrow R_n)$$

One can also represent a "stack discipline" for resources, as follows:

$$x \Leftarrow (R_1 \mid \cdots x \Leftarrow (R_{n-1} \mid x \Leftarrow R_n))$$

Regarding the case of π within $\pi^\star$, some π-terms which are usually considered as equal, like $\mathsf{rec}\, r.(\lambda x)P$ and $\mathsf{0}$, where $\mathsf{0} =_{\mathrm{def}} (\nu u)u$, are obviously different in $\pi^\star$ (the first one converges), but the case of well-sorted π-terms (see below) deserves further investigations.

4. Typing the Blue Calculus

Since everything written in the λ or π calculi may be read in blue style, we get a very rich calculus. One usually imposes some discipline on the use of higher-order values, or channel names in the case of π, to get some control over the programs one is allowed to run. In this section and the following one, we show that, not only $\pi^\star$ contains both λ and π, but there is a type discipline that encompasses both the simply typed λ-calculus and the well-sorted π-calculus. The types are the same as for the simply typed λ-calculus, namely

$$\tau ::= t \mid (\tau \to \tau)$$

where t is any type variable or constant. The (simple) type system $\mathcal{S}$ deals with sequents $\Gamma \vdash P : \tau$ where the assumption Γ is a mapping from a finite set of names to types. We denote by $\Gamma_{\lfloor u}$ the mapping obtained from Γ by removing u from its domain. As usual $u : \tau, \Gamma$ is the assumption Δ such that $\Delta(u) = \tau$ and $\Delta_{\lfloor u} = \Gamma$. The rules of the system are given in Table 3 (in this system we implicitly assume that we are dealing with terms up to α-conversion, otherwise we could not be able to type $(\lambda x)(\lambda x)x$ for instance).

The rules for declarations deserve some comment: a sequent $u:\sigma, \Gamma \vdash P:\tau$ means that if u is used in P, it is with the type σ. Therefore, if we have a term Q of appropriate type, that is $Q:\sigma$, we can use it as a resource for u in P, that is, $(P \mid \langle u \Leftarrow Q\rangle)$ should have type τ. This may be inferred using the rule for parallel composition, since a declaration for u may be assigned any type τ, provided the context Γ declares u of the same type as its declared value. Indeed, the following is a proof in the system $\mathcal{S}$:

$$\frac{\dfrac{\vdots}{\Gamma \vdash P:\tau} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash Q:\sigma}}{\Gamma \vdash \langle u \Leftarrow Q\rangle:\tau}\ \Gamma(u)=\sigma}{\Gamma \vdash (P \mid \langle u \Leftarrow Q\rangle):\tau}$$

and similarly for $(P \mid \langle u = Q\rangle)$. The rule for parallel composition may be understood as follows: the assumption Γ in a sequent $\Gamma \vdash S:\tau$ provides some information on free names that is shared by the components of the "distributed system" S. Then, in particular, the components of $S = (P \mid Q)$ must agree on this information.

We denote by $\Gamma \vdash P:\tau\ [\mathcal{S}]$ the fact that this sequent may be inferred using the rules of the system $\mathcal{S}$. The following property, stating in particular that weakening is valid in $\mathcal{S}$, is standard.

LEMMA 4.1.

$u \notin \mathsf{fn}(P) \ \Rightarrow\ \Gamma \vdash P:\tau\ [\mathcal{S}] \ \Leftrightarrow\ u:\sigma, \Gamma \vdash P:\tau\ [\mathcal{S}]$

Our typing system $\mathcal{S}$ extends in a direct manner Curry's type system for the λ-calculus. Let us recall that the latter has the same types, and the following inference rules - still using Γ to denote an assumption that holds for variables from $\mathcal{X}$:

$$\frac{}{\Gamma \vdash x:\tau}\ \Gamma(x)=\tau \qquad \frac{x:\sigma, \Gamma \vdash M:\tau}{\Gamma \vdash \lambda x M:\sigma\to\tau}$$

$$\frac{\Gamma \vdash M:\sigma\to\tau\ ,\ \Gamma \vdash N:\sigma}{\Gamma \vdash MN:\tau}$$

We write $\Gamma \vdash M:\tau\ [\mathcal{C}]$ to mean that this sequent is provable using Curry's system. We recall that $[\![\cdot]\!]$ denotes the translation from λ to π^*. Then we have:

PROPOSITION 4.2. *If $\Gamma \vdash M:\tau\ [\mathcal{C}]$ then $\Gamma \vdash [\![M]\!]:\tau\ [\mathcal{S}]$. Conversely if $\Gamma \vdash [\![M]\!]:\tau\ [\mathcal{S}]$ then $\Delta \vdash M:\tau\ [\mathcal{C}]$ with $\Delta = \Gamma_{\lfloor v_1,\ldots,v_k}$ where $v_1,\ldots,v_k$ are the names occurring in Γ and not in M.*

The proof, using the Lemma 4.1, is easy. One can also see that the optimized version of the translation, where $[\![Mx]\!]$ is $[\![M]\!]x$, also preserves the typing.

Given this result, we may use the ordinary typing rule for $(PQ) =_{\mathrm{def}} (\nu v)(Pv \mid v = Q)$ where v is fresh. For instance, the following inference holds in $\mathcal{S}$:

$$\frac{\dfrac{\dfrac{\vdots}{f:\sigma\to\sigma,\ \mathsf{fix}:(\sigma\to\sigma)\to\sigma \vdash f(\mathsf{fix}f):\sigma}}{\mathsf{fix}:(\sigma\to\sigma)\to\sigma \vdash (\lambda f)f(\mathsf{fix}f):(\sigma\to\sigma)\to\sigma}}{\mathsf{fix}:(\sigma\to\sigma)\to\sigma \vdash \langle \mathsf{fix}=(\lambda f)f(\mathsf{fix}f)\rangle:\tau}$$

Similarly, one can use a derived $(\mathsf{def}\ x = R\ \mathsf{in}\ P)$ construct, that stands for $(\nu x)(P \mid x = R)$, together with the typing:

$$\frac{\Delta \vdash R:\sigma \quad , \quad x:\sigma, \Gamma \vdash P:\tau}{\Gamma \vdash (\mathsf{def}\ x = R\ \mathsf{in}\ P):\tau}\ (*)$$

$(*)$ where $\Delta = \Gamma$ or $\Delta = x:\sigma, \Gamma$. Then, denoting I the identity $(\lambda x)x$, one could prove for instance that the term $(\mathsf{def}\ \mathsf{fix} = (\lambda f)f(\mathsf{fix}f)\ \mathsf{in}\ (\mathsf{fix}\,\mathsf{I}))$ has any type. Since the system $\mathcal{S}$ can handle recursion directly, it does not enjoy the strong normalization property. A simple counter-example is provided by $\Omega = (\nu u)(u \mid u = u)$ that can be given any type. Another fact is that typability does not imply confluence. For instance, the reader may check that $\vdash \oplus:\tau\to\tau\to\tau$, and that typability does not preclude the non-determinism due to transmission delays, as in $(uv_1\cdots v_k \mid uw_1\cdots w_k \mid \langle u \Leftarrow R\rangle)$.

Nevertheless, the system $\mathcal{S}$ has the desirable property that types are preserved during evaluation. That is, our typing system enjoys the well-known "subject reduction property". To show this point, we first prove that typing is preserved by structural manipulations. A preliminary observation is that, since there is exactly one rule for each construct of the calculus, the relation "to have the same types" is obvioulsy a congruence (which contains α-conversion).

LEMMA 4.3. $\Gamma \vdash P:\tau\ \&\ Q \equiv P\ \Rightarrow\ \Gamma \vdash Q:\tau$

PROOF: we proceed by induction on the definition of $\equiv$. The proof is straightforward, therefore we only examine some cases, leaving the remaining ones to the reader.

(1) $(P_0 \mid P_1)u \equiv (P_0u \mid P_1u)$ and $\Gamma \vdash (P_0u \mid P_1u):\tau\ [\mathcal{S}]$.

This sequent can only be proved as follows:

$$\frac{\Pi_0 \quad , \quad \Pi_1}{\Gamma \vdash (P_0u \mid P_1u):\tau}$$

with

$$\Pi_i = \frac{\dfrac{\vdots}{\Gamma \vdash P_i:\sigma_i\to\tau} \qquad \dfrac{}{\Gamma \vdash u:\sigma_i}\ \Gamma(u)=\sigma_i}{\Gamma \vdash P_iu:\tau}$$

Since $\sigma_0 = \Gamma(u) = \sigma_1$, we have $\Gamma \vdash (P_0 \mid P_1):\sigma_i\to\tau$, hence $\Gamma \vdash (P_0 \mid P_1)u:\tau$.

(2) $\langle u = P\rangle \equiv Q$ where $Q = \langle u \Leftarrow (P \mid \langle u = P\rangle)\rangle$, and $\Gamma \vdash Q:\tau$. This must be inferred as follows:

$$\frac{\dfrac{\dfrac{\vdots}{\Gamma \vdash P:\sigma} \qquad \dfrac{\vdots}{\Gamma \vdash \langle u = P\rangle:\sigma}}{\Gamma \vdash (P \mid \langle u = P\rangle):\sigma}\ \Gamma(u)=\sigma}{\Gamma \vdash \langle u \Leftarrow (P \mid \langle u = P\rangle)\rangle:\tau}$$

syntax:

$$P ::= \bar{u}\tilde{v} \mid u(\tilde{v})P \mid !u(\tilde{v})P \mid (P \mid P) \mid (\nu u)P$$

structural equivalence:

$$(P \mid Q) \equiv (Q \mid P)$$
$$((P \mid Q) \mid R) \equiv (P \mid (Q \mid R))$$
$$((\nu u)P \mid Q) \equiv (\nu u)(P \mid Q) \qquad (u \text{ not free in } Q)$$

reduction:

$$(\bar{u}\tilde{w} \mid u(\tilde{v})P) \to [\tilde{w}/\tilde{v}]P$$
$$(\bar{u}\tilde{w} \mid !u(\tilde{v})P) \to ([\tilde{w}/\tilde{v}]P \mid !u(\tilde{v})P)$$
$$P \to P' \Rightarrow (P \mid Q) \to (P' \mid Q)$$
$$P \to P' \Rightarrow (\nu u)P \to (\nu u)P'$$
$$P \to P' \;\&\; Q \equiv P \Rightarrow Q \to P'$$

Table 4: the π-Calculus

hence $\Gamma \vdash \langle u = P \rangle : \tau$ $[\mathcal{S}]$ since $\Gamma \vdash P : \sigma$ $[\mathcal{S}]$ and $\Gamma(u) = \sigma$ ❑

One may remark that the "sharing rule" $(P \mid Q)u \equiv (Pu \mid Qu)$ seems to leave little room for a possible polymorphic extension of $\mathcal{S}$: a name can only be used polymorphically in the head position, not as an argument. However, one should not forget that this structural manipulation is mainly used from left to right.

PROPOSITION (SUBJECT REDUCTION) 4.4.
$\Gamma \vdash P : \tau \;\&\; P \to P' \Rightarrow \Gamma \vdash P' : \tau$

PROOF: by induction on the inference of $P \to P'$. If this is proved using a context rule, then we use the fact that the proof of $\Gamma \vdash P : \tau$ may be decomposed according to the structure of P, and we use the induction hypothesis. In the case where we use structural transformations on P to prove $P \to P'$, then we use the previous lemma and the induction hypothesis. We are then left with the cases of (β) and ϱ.
(1) If $P = (\lambda u)R)v$ and $P' = [v/u]R$, then $\Gamma \vdash (\lambda u)R)v : \tau$ can only be inferred from $u : \sigma$, $\Gamma \vdash R : \tau$ $[\mathcal{S}]$, with $\Gamma(v) = \sigma$. It is easy to check that this implies $\Gamma \vdash [v/u]R : \tau$ $[\mathcal{S}]$.
(2) If $P = (u \mid \langle u \Leftarrow P' \rangle)$ then the proof of $\Gamma \vdash P : \tau$ must be

$$\dfrac{\Gamma(u) = \tau \quad \dfrac{}{\Gamma \vdash u : \tau} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash P' : \sigma}}{\Gamma \vdash \langle u \Leftarrow P' \rangle : \tau} \quad \Gamma(u) = \sigma}{\Gamma \vdash (u \mid \langle u \Leftarrow P' \rangle) : \tau}$$

therefore $\Gamma \vdash P' : \tau$ $[\mathcal{S}]$ since $\sigma = \Gamma(u) = \tau$ ❑

The proof of the subject reduction property is remarkably simple. This is because the granularity of reduction in the blue calculus is quite fine. One should notice that to get this property, it is required that in the typing of $\langle u \Leftarrow R \rangle$ (and of $\langle u = R \rangle$), R has the *same* type as the one declared for u: we cannot predict which resource will be used for a particular occurrence of the name u, therefore to ensure a correct typing, these resources must all have the same type.

5. The π-Calculus

We assume that π-calculus names are also names for π^*. We summarize in Table 4 the syntax and reduction of the (asynchronous) version of the π-calculus that we use, where $\tilde{u}$ denotes a tuple of names (we have omitted the rule that structural transformations may be used in parallel and restriction contexts). In the communication rules, $\tilde{w}$ and $\tilde{v}$ are supposed to be tuples of names of the same length. In Section 2 we sketched a translation from π to the blue calculus. Here it is:

$$[\![\bar{u}v_1 \cdots v_k]\!] = uv_1 \cdots v_k$$
$$[\![u(v_1, \ldots, v_k)P]\!] = \langle u \Leftarrow (\lambda v_1 \ldots v_k)[\![P]\!] \rangle$$
$$[\![!u(v_1, \ldots, v_k)P]\!] = \langle u = (\lambda v_1 \ldots v_k)[\![P]\!] \rangle$$
$$[\![P \mid Q]\!] = ([\![P]\!] \mid [\![Q]\!])$$
$$[\![(\nu u)P]\!] = (\nu u)[\![P]\!]$$

It should be clear that reductions in the π-calculus are mimicked in the following way:

LEMMA 5.1. *Let P be a π-term.*
If $P \to_\pi P'$ then $[\![P]\!] \to_\varrho \stackrel{*}{\to}_{(\beta)} [\![P']\!]$.

For instance one has

$$\begin{aligned}[\![\bar{u}\tilde{w} \mid u(\tilde{v})P]\!] &= (u\tilde{w} \mid u \Leftarrow (\lambda\tilde{v})[\![P]\!]) \\ &\equiv (u \mid u \Leftarrow (\lambda\tilde{v})[\![P]\!])\tilde{w} \\ &\to_\varrho ((\lambda\tilde{v})[\![P]\!])\tilde{w} \\ &\stackrel{*}{\to}_{(\beta)} [\tilde{w}/\tilde{v}][\![P]\!] \equiv [\![[\tilde{w}/\tilde{v}]P]\!]\end{aligned}$$

This shows that even if the π-calculus is a "name-passing" calculus, it is also in a somewhat hidden manner a *higher-order* calculus: in the head position, that is as the channel on which a message is sent, a name stands for a process, or more generally for an abstraction. It should also be clear that, conversely, reductions of $[\![P]\!]$ in the blue calculus do not always correspond to reductions of P in π – in the example above, $\tilde{v}$ and $\tilde{w}$ need not have the same length for a reduction to occur. However, an exact operational correspondence between π-terms and their translation holds for well-sorted terms.

$$\widetilde{v}:\widetilde{\zeta},\, u:\mathsf{Ch}(\widetilde{\zeta}),\, \Sigma \succ \bar{u}\widetilde{v} \qquad \frac{\widetilde{v}:\widetilde{\zeta},\, u:\mathsf{Ch}(\widetilde{\zeta}),\, \Sigma \succ P}{u:\mathsf{Ch}(\widetilde{\zeta}),\, \Sigma \succ u(\widetilde{v})P} \qquad \frac{\widetilde{v}:\widetilde{\zeta},\, u:\mathsf{Ch}(\widetilde{\zeta}),\, \Sigma \succ P}{u:\mathsf{Ch}(\widetilde{\zeta}),\, \Sigma \succ\, !u(\widetilde{v})P}$$

$$\frac{\Sigma \succ P\,,\, \Sigma \succ Q}{\Sigma \succ (P \mid Q)} \qquad \frac{\Sigma \succ P}{\Sigma_{\lfloor u} \succ (\nu u)P}$$

Table 5: Sorting the π-Calculus

As for the sorting, we deal with a simple version of Milner's system [18], where sorts are not recursive. Such a system has been considered by Vasconcelos and Honda [29], and by Turner in his thesis [28]. The *sorts* are built as follows:

$$\zeta ::= t \mid \mathsf{Ch}(\zeta_1, \ldots, \zeta_n)$$

The sort $\mathsf{Ch}(\zeta_1, \ldots, \zeta_n)$ is that of a channel that carries a tuple of names of sorts $\zeta_1, \ldots, \zeta_n$. The (simple) sorting system deals with judgements of the form $\Sigma \succ P$, where P is a π-term and Σ is a sort assumption, that is a mapping from a finite set of names to sorts. We read $\Sigma \succ P$ as "P is well-sorted under the assumption Σ", or respects Σ. This is also sometimes written $\Sigma \vdash P : o$, or $\Sigma \vdash P : ok$. Turner's system, adapted to our asynchronous version of the π-calculus, is recalled in Table 5. The following lemma, together with the preceding one, states that for well-sorted processes, π-calculus reduction $\rightarrow_\pi$ is essentially the same as reduction in the blue calculus:

LEMMA 5.2. *Let P be a π-term. If P is well-sorted, that is $\Sigma \succ P$ for some Σ, and $[\![P]\!] \rightarrow Q$ then $[\![P]\!] \rightarrow_\varrho Q$ and there exists P' such that $P \rightarrow_\pi P'$ and $[\![P']\!] \equiv \mathsf{nf}_\beta(Q)$.*

The (easy) proof is omitted. To relate the sorting of π-terms with the typing of their interpretation in $\pi^\star$, let us assume given a type o (or ok). Then we translate the sorts into types as follows:

$$[\![t]\!] = t$$
$$[\![\mathsf{Ch}(\zeta_1, \ldots, \zeta_k)]\!] = ([\![\zeta_1]\!] \rightarrow \cdots ([\![\zeta_k]\!] \rightarrow o) \cdots)$$

For instance, the translation of the sort $\mathsf{Ch}(\mathsf{Ch}(), \mathsf{Ch}())$ of the "boolean channels" in the π-calculus (see [28]) is $o \rightarrow o \rightarrow o$. A sorting assumption Σ is translated into a typing assumption $[\![\Sigma]\!]$ in the obvious way. A fact is that sorting is a particular case of typing:

PROPOSITION 5.3. *If $\Sigma \succ P$ then $[\![\Sigma]\!] \vdash [\![P]\!] : o$ [$\mathcal{S}$].*

In [18,28] a rule is given regarding the well-sortedness of a replicated process $!P$. If we define $!P$ as $\mathsf{rec}\, r.(P \mid r)$, that is $(\nu r)(r \mid r = (P \mid r))$ where $r \notin \mathsf{fn}(P)$, then it is easy to see that $!P$ has type τ whenever P has type τ, thus generalising the sorting rule.

Clearly one can give types to π-terms that are more general than sortings. Then the blue calculus appears to be more flexible than the π-calculus. For instance, a recursive agent $\mathsf{rec}\, r.(\lambda x)P = [\![(\nu r)(\bar{r} \mid !r(x)P)]\!]$ cannot be well-sorted in the π-calculus, while in $\pi^\star$ it has the type of $(\lambda x)P$. Another example is provided by the typing of the "RPC-like" facility of PICT, denoted (let $x_1, \ldots, x_n = f(a_1, \ldots, a_m)$ in P) by Turner [28], which is a notation for $(\nu r)(f a_1 \cdots a_m r \mid r \Leftarrow (\lambda x_1 \ldots x_n)P)$. In our calculus, this may be given the type τ, provided P has type σ under the assumption $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, and the environment supplies f with type:

$$f : \tau_1 \rightarrow \cdots \tau_m \rightarrow (\sigma_1 \rightarrow \cdots \sigma_n \rightarrow \sigma) \rightarrow \tau$$

and the arguments a_i have type τ_i. In the π-calculus, we may only type this term if $\sigma = o = \tau$. A particular case of this "RPC-like" facility is the *synchronous* send $\bar{s}v_1 \cdots v_k.P$ of the original π-calculus [16,18], which may be regarded as an abbreviation for

$$(\nu r)(s v_1 \cdots v_k r \mid r \Leftarrow P) \qquad (r \text{ fresh})$$

together with the synchronous input $s(x_1, \ldots, x_k).Q$, which is

$$s \Leftarrow (\lambda x_1 \ldots x_k)(\lambda r)(Q \mid r) \qquad (r \text{ fresh})$$

As we noted in [5], synchronous message passing has the flavour of a process passing construct, where the continuation P is passed in an output $\bar{s}v_1 \cdots v_k.P$ – this is clear if we write it as $s v_1 \cdots v_k P$. The encoding of the synchronous π-calculus into the blue calculus we just sketched is also well-behaved with respect to sorts (as given in [28]) and types, though with a different translation $[\![.]\!]'$, mapping $\mathsf{Ch}(\zeta_1, \ldots, \zeta_k)$ onto $[\![\zeta_1]\!]' \rightarrow \cdots [\![\zeta_k]\!]' \rightarrow o \rightarrow o$.

6. CPS Transforms

The technique of continuations is widely used in the semantics of functional programing languages and control operators. Here we are specifically interested into *continuation passing style transformations*, that map programs written in one language into another – though by "language" one should understand "evaluation strategy" rather than syntax. The best known of these transforms is the simulation of call-by-value by call-by-name in the λ-calculus, studied by Plotkin in [21] (see [27] for earlier references). If we compose this simulation with the translation (optimized or not) from λ into $\pi^\star$ given in Section 2, we get the following mapping, where we use k, h, $j \ldots$ for "continuations":

$$\begin{aligned}
[\![x]\!]^\bullet &= (\lambda k)(\nu v)(kv \mid \langle v = x\rangle) \qquad \text{or} \\
&= (\lambda k)kx \\
[\![\lambda x M]\!]^\bullet &= (\lambda k)(\nu v)(kv \mid \langle v = (\lambda x)[\![M]\!]^\bullet\rangle) \\
[\![MN]\!]^\bullet &= (\lambda k)(\nu v)([\![M]\!]^\bullet v \mid \langle v = (\lambda h)(\nu w)([\![N]\!]^\bullet w \\
&\qquad\qquad \mid \langle w = (\lambda j)hjk\rangle)))
\end{aligned}$$

To state the correctness of this transformation, one actually needs to compose it with the application to a "continuation" argument, thus dealing with $[\![M]\!]^\bullet k$. Now if we perform the following operations:

(i) (β)-normalize $[\![M]\!]^\bullet k$,

(ii) make an η-expansion on $[\![M]\!]^\bullet$ in $[\![\lambda x M]\!]^\bullet$ (which is harmless, since $[\![M]\!]^\bullet$ is always an abstraction), and also on x in $[\![x]\!]^\bullet$ (though this is optional)

then we get:

$$\begin{aligned}
[\![x]\!]^\bullet k &\equiv_{(\beta)\eta} (\nu v)(kv \mid \langle v = (\lambda h)xh\rangle) \qquad \text{or} \\
&\equiv_{(\beta)} kx \\
[\![\lambda x M]\!]^\bullet k &\equiv_{(\beta)\eta} (\nu v)(kv \mid \langle v = (\lambda x)(\lambda h)[\![M]\!]^\bullet h\rangle) \\
[\![MN]\!]^\bullet k &\equiv_{(\beta)\eta} (\nu v)([\![M]\!]^\bullet v \mid \langle v = (\lambda h)(\nu w)([\![N]\!]^\bullet w \\
&\qquad\qquad \mid \langle w = (\lambda j)hjk\rangle)))
\end{aligned}$$

One can see that these are terms written in the π-calculus, provided that we regard $[\![M]\!]k$ as a meta-application. Furthermore, this transformation is strikingly close to the encodings for the call-by-value λ-calculus into the π-calculus one finds in the literature. Milner in the preliminary version of [17] considered the two encodings, though written in the monadic, synchronous π-calculus, and allowing to compute $[\![N]\!]^\bullet w$ concurrently with $[\![M]\!]^\bullet v$ in $[\![MN]\!]^\bullet u$ (see [28] and [2] for the "optimized" version we presented here, further slightly improved by replacing $\langle v = \cdots\rangle$ with $\langle v \Leftarrow \cdots\rangle$ in $[\![MN]\!]^\bullet u$).

Regarding types and sorts, we can use a remark of Meyer and Wand [15]. They noticed that there is a transformation on types that parallels the CPS transform of call-by-value into call-by-name, which is:

$$\begin{aligned}
t^\bullet &= t \\
(\sigma \to \tau)^\bullet &= \sigma^\bullet \to ((\tau^\bullet \to o) \to o)
\end{aligned}$$

We may also denote $(\tau \to o)$ as $\neg^o\tau$ (see [19]). Then, given our Proposition 4.2, we can reformulate Meyer and Wand's remark as follows, denoting by $\Gamma^\bullet$ the assumption that is the composition of Γ with $[\![\cdot]\!]^\bullet$:

LEMMA 6.1. $\Gamma \vdash M : \tau\ [\mathcal{C}] \ \Rightarrow\ \Gamma^\bullet \vdash [\![M]\!]^\bullet : \neg^o\neg^o\tau^\bullet\ [\mathcal{S}] \ \Rightarrow\ k : \neg^o\tau^\bullet,\ \Gamma^\bullet \vdash [\![M]\!]^\bullet k : o\ [\mathcal{S}]$

Notice that, given our previous interpretation of sorts as types, the $^\bullet$ transformation on types is formally the same as a transformation of types into sorts (see [2]):

$$\begin{aligned}
t^\bullet &= [\![t]\!] \\
(\sigma \to \tau)^\bullet &= [\![\mathsf{Ch}(\sigma^\bullet, \mathsf{Ch}(\tau^\bullet))]\!]
\end{aligned}$$

Then, considering $[\![M]\!]^\bullet k$ as a π-term, and viewing a continuation as a monadic channel, this lemma is the same as Turner's Proposition 6.13 [28] relating the types of call-by-value λ-terms with the sorts of their encoding (all the encodings considered by Turner in the Chapter 6 of his Thesis [28] actually use the asynchronous π-calculus as a target).

The CPS transform of call-by-name into call-by-value, also studied by Plotkin in [21], is less well-known. Written in blue style, it provides us with another encoding of λ into π^*. This may actually be extended to the whole blue calculus, as a function $[\![\cdot]\!]^\circ$ that may be regarded as mapping π^* into $\mathcal{N} \to \pi$, thus justifying our claim that π^* is "the π-calculus in direct style". This mapping is given as follows, where k is a variable standing for "continuation":

$$\begin{aligned}
[\![u]\!]^\circ &= (\lambda k)uk \\
[\![(\lambda x)P]\!]^\circ &= (\lambda k)(\nu p)(kp \mid \langle p = (\lambda x)[\![P]\!]^\circ\rangle) \\
[\![Pu]\!]^\circ &= (\lambda k)(\nu p)([\![P]\!]^\circ p \mid \langle p = (\lambda z)zuk\rangle) \\
[\![P \mid Q]\!]^\circ &= ([\![P]\!]^\circ \mid [\![Q]\!]^\circ) \\
[\![(\nu u)P]\!]^\circ &= (\nu u)[\![P]\!]^\circ \\
[\![r \Leftarrow P]\!]^\circ &= (\lambda k)\langle r \Leftarrow [\![P]\!]^\circ\rangle \\
[\![r = P]\!]^\circ &= (\lambda k)\langle r = [\![P]\!]^\circ\rangle
\end{aligned}$$

where the introduced names k, z and p are fresh (and distinct). One can check that, as far as λ-terms are concerned, this is indeed Plotkin's simulation of call-by-name by call-by-value, written in blue style (with an η-expansion on the translation of variables, but in the call-by-value setting, a variable x is a value, therefore η-conversion on x is valid, i.e. $x =_v \lambda k.xk$). Furthermore, it can be seen that, up to the same manipulations as above, $[\![M]\!]^\circ k$ corresponds to an encoding of the call-by-name λ-calculus into the π-calculus given by Ostheimer and Davie (see [28]. The connection with CPS is mentioned by Fournet and Gonthier in [11], where they give essentially the same encoding). More generally, $[\![P]\!]^\circ k$ may be regarded as a π-term, up to innocuous η-expansion.

Before discussing the correctness of this CPS transform, we first examine how types are correspondingly transformed (*cf.* [13,19]). Recall that we assumed given a type o, and that we denoted $(\tau \to o)$ by $\neg^o\tau$. Then the $(\cdot)^\circ$ translation on types is:

$$\begin{aligned}
t^\circ &= \neg^o\neg^o t \\
(\sigma \to \tau)^\circ &= \neg^o\neg^o(\sigma^\circ \to \tau^\circ)
\end{aligned}$$

By definition, for any τ there exists a ς such that $\tau^\circ = \neg^o\neg^o\varsigma$. The following lemma extends Murthy's observation on call-by-name translation [19] and Turner's Proposition 6.18 [28].

LEMMA 6.2. $\Gamma \vdash P : \tau\ [\mathcal{S}] \ \Rightarrow\ \Gamma^\circ \vdash [\![P]\!]^\circ : \tau^\circ\ [\mathcal{S}] \ \Rightarrow\ k : \neg^o\varsigma,\ \Gamma^\circ \vdash [\![P]\!]^\circ k : o\ [\mathcal{S}]$ *where* $\tau^\circ = \neg^o\neg^o\varsigma$.

The (straightforward) proof is omitted.

The CPS transform $[\![\cdot]\!]^\circ$ is only correct if we impose a restriction, namely that a *declared name should not be abstracted*. Typically, a term like $(\lambda u)\langle u \Leftarrow R\rangle$ does not fulfil this requirement. The only example I know that disobeys this discipline, in a polyadic calculus, is Milner's encoding of the call-by-name λ-calculus [18] that we recalled in the introduction – and that we do not need anymore, since $\lambda \subset \pi^*$ in a direct manner. Moreover, in a recent paper [10] Boreale has shown that one can embed the full π-calculus into the subset of terms satisfying the restriction.

The reason why we need to have this distinction between variables and references is interesting: we have to ensure a well-known "property of replication" (see [18,23]), which is, roughly, that a local declaration (def $r = R$ in $\cdots$) may be distributed over parallel components, provided that r is not defined elsewhere. Moreover, the restricted calculus is better-behaved in many other respects (see the conclusion).

Instead of considering only a part of the set of terms, we introduce a restricted syntax, assuming that the set $\mathcal{N}$ of names is the disjoint union of $\mathcal{X}$, the denumerable set of *variables* x, y, $z\ldots$, and $\mathcal{R}$, the denumerable set of *references* p, q, $r\ldots$ and that only variables may be abstracted, while only references may be declared (a similar restriction is adopted in Oz [26], and also in the JOIN calculus [11], where a declared name is bound by a restriction). Therefore the grammar is:

$$\begin{array}{lll} u ::= x \mid r & & \textit{names} \\ P ::= A \mid D \mid (P \mid P) \mid (\nu u)P & & \textit{processes} \\ A ::= u \mid (\lambda x)P \mid (Pu) & & \textit{agents} \\ D ::= \langle r \Leftarrow P\rangle \mid \langle r = P\rangle & & \textit{declarations} \end{array}$$

The substitutions are assumed to respect the distinction between names, in the sense that a reference can only be renamed for α-conversion purposes. For terms P, Q written in this syntax, we have the following adequacy result:

THEOREM 6.3. $k \notin \mathsf{fn}(P \mid Q)$ & $[\![P]\!]^\circ k \simeq [\![Q]\!]^\circ k \;\Rightarrow\; P \simeq Q$

For lack of space, we only provide a sketch of the proof.

(1) A first step is to give to the CPS a technically more convenient form. To this end, we notice that η-expansion is valid on translated terms:

LEMMA 6.4. $j \notin \mathsf{fn}(P) \;\Rightarrow\; (\lambda j)([\![P]\!]^\circ j) \simeq [\![P]\!]^\circ$

The proof, by induction on P, uses the Exercise 3.4 and the Lemma 3.2. Now let us define the mapping $\mathcal{K}$: $\pi^\star \times \mathcal{N} \to \pi^\star$ as follows:

$$\begin{array}{rcl} \mathcal{K}(u,k) & = & uk \\ \mathcal{K}((\lambda x)P,k) & = & (\nu p)(kp \mid \langle p = (\lambda x)(\lambda j)\mathcal{K}(P,j)\rangle) \\ \mathcal{K}(Pu,k) & = & (\nu p)(\mathcal{K}(P,p) \mid \langle p = (\lambda z)zuk\rangle) \\ \mathcal{K}(P \mid Q,k) & = & (\mathcal{K}(P,k) \mid \mathcal{K}(Q,k)) \\ \mathcal{K}((\nu u)P,k) & = & (\nu u)\mathcal{K}(P,k) \\ \mathcal{K}(r \Leftarrow P,k) & = & \langle r \Leftarrow (\lambda j)\mathcal{K}(P,j)\rangle \\ \mathcal{K}(r = P,k) & = & \langle r = (\lambda j)\mathcal{K}(P,j)\rangle \end{array}$$

An immediate consequence of the Lemma 3.2 and of the previous lemma is:

COROLLARY 6.5. $k \notin \mathsf{fn}(P) \;\Rightarrow\; \mathcal{K}(P,k) \simeq [\![P]\!]^\circ k$

Then to prove the Theorem amounts to show:

$$k \notin \mathsf{fn}(P \mid Q) \;\&\; \mathcal{K}(P,k) \simeq \mathcal{K}(Q,k) \;\Rightarrow\; P \simeq Q$$

(2) Our improved CPS transform $\mathcal{K}$ almost preserves structural equivalence, but not quite. We need to take into account two new laws: one is that inaccessible resources may be garbage collected, the other is that shared inexhaustible resources may be cloned. To state this more formally, we denote by $\mathsf{decl}(P) \subseteq \mathcal{R}$ the set of (free) *declared* names of P – defined in the obvious way, that is $\mathsf{decl}(r \Leftarrow P) = \{r\} \cup \mathsf{decl}(P)$, $\mathsf{decl}((\nu r)P) = \mathsf{decl}(P) - \{r\}$, and so on. It is easy to see that, in our restricted calculus, evaluation cannot create declared names, since $\mathsf{decl}((\lambda x)P)r) = \mathsf{decl}(P)$, that is:

LEMMA 6.6. $P \to P' \;\Rightarrow\; \mathsf{decl}(P') \subseteq \mathsf{decl}(P)$

Now let us define $\cong$ as the least equivalence that satisfies the same laws as $\equiv$, plus the following two – recall that we denoted $(\nu r)(P \mid r = R)$ by $(\mathsf{def}\ x = R\ \mathsf{in}\ P)$:

$$\boxed{\begin{array}{rl} (\nu r)(P \mid \langle r \Leftarrow R\rangle) \cong P & (*) \\ (\mathsf{def}\ r = R\ \mathsf{in}\ P \mid Q) \cong (\mathsf{def}\ r = R\ \mathsf{in}\ P) \mid & \\ \qquad (\mathsf{def}\ r = R\ \mathsf{in}\ Q) & (**) \end{array}}$$

$(*)$ r not free in P

$(**)$ $r \notin \mathsf{decl}(P \mid Q \mid R)$

The second law is a well-known "property of replication" (see [18]. Our syntactic requirement is slightly more liberal than the hypothesis required in [18], since here a declared name, i.e. an "input channel", can be passed as argument). These two laws are quite natural – for instance the first one could be incorporated into the operational semantics. We can then state how structural equivalence is transformed by the CPS:

LEMMA 6.7. $k \notin \mathsf{fn}(P)$ & $P \equiv Q \;\Rightarrow\; \mathcal{K}(P,k) \cong \mathcal{K}(Q,k)$

(3) Now we aim at establishing an operational correspondence between P and $\mathcal{K}(P,k)$. Roughly, we would like to show that $\mathcal{K}(P,k)$ converges exactly whenever P converges. Let us first examine how the reductions of P are transformed. As usual, we have to manage "administrative reductions" (see [21]). These are either (β)-reductions, or fetching an inexhaustible resource for a name which is uniquely (and locally) declared. That is, if we define the reduction relation $P \triangleright P'$ as follows:

$P \triangleright P'$ if and only if either $P \to_{(\beta)} P'$ or
$P \equiv (\nu u_1 \ldots u_n)(u_i v_1 \cdots v_k \mid \langle u_i = R\rangle \mid S)$, with
$i \in \{1,\ldots,n\}$ and $u_i \notin \mathsf{decl}(S)$, and
$P' \equiv (\nu u_1 \ldots u_n)(Rv_1 \cdots v_k \mid \langle u_i = R\rangle \mid S)$.

then administrative reductions are particular cases of $P \triangleright P'$. Clearly $P \triangleright P' \;\Rightarrow\; \mathbf{E}[P] \triangleright \mathbf{E}[P']$ for any evaluation context $\mathbf{E}$. Moreover, these reductions commute with any other ones:

LEMMA 6.8.

$P \triangleright P_0$ & $P \to P_1 \;\Rightarrow\; P_1 \equiv P_0$ or $\exists P'.\ P_1 \triangleright P'$ & $P_0 \to P'$

Let us denote by $\sim$ the equivalence generated by $\triangleright \cup \cong$. A first part of the operational correspondence concerns (β)-reduction:

LEMMA 6.9. $P \to_{(\beta)} P' \;\Rightarrow\; \mathcal{K}(P,k) \sim \mathcal{K}(P',k)$

In particular, if P is converted into a canonical form Q:

$$\begin{array}{rcl} P \equiv_{(\beta)} Q = (\nu u_1 \ldots u_n)(V_1 \mid \cdots \mid V_m \mid & & \\ \qquad M_1 \mid \cdots \mid M_s \mid & & \\ \qquad r_1 \Leftarrow R_1 \mid \cdots \mid r_k \Leftarrow R_k) & & \end{array}$$

then $\mathcal{K}(P,k) \sim \mathcal{K}(Q,k)$. This allows us to prove another part of the operational correspondence, regarding resource fetching:

LEMMA 6.10. $P \to_\varrho P' \;\Rightarrow\; \mathcal{K}(P,k) \sim\!\!\to_\varrho\!\sim \mathcal{K}(P',k)$

To establish the converse correspondence, we use the fact that $\sim$ is a kind of bisimulation:

LEMMA 6.11.

(i) $P \sim Q$ & $P \to P' \;\Rightarrow\; P' \sim Q$ or $\exists Q'.\ Q \to Q'$ & $P' \sim Q'$

(ii) $P \sim Q$ & $P{\Downarrow} \;\Rightarrow\; Q{\Downarrow}$

Assume that $\mathcal{K}(P,k)$ performs some reduction, and that P has a canonical form Q as above. Then, since $\mathcal{K}(P,k) \sim \mathcal{K}(Q,k)$, the reduction from $\mathcal{K}(P,k)$ is either "absorbed" in this transformation, or still may be performed from $\mathcal{K}(Q,k)$. That is, we have:

LEMMA 6.12.
$\mathcal{K}(P,k) \to S \;\Rightarrow\; S \sim \mathcal{K}(P,k)$ or $\exists P'.\ P \to P'\ \&\ S \sim \mathcal{K}(P',k)$

In particular, this implies:

$$\mathcal{K}(P,k) \stackrel{*}{\to} S \;\Rightarrow\; \exists P'.\ P \stackrel{*}{\to} P'\ \&\ S \sim \mathcal{K}(P',k)$$

Finally we can prove the main lemma, relating the convergence of P with that of $\mathcal{K}(P,k)$. Notice that an abstraction is transformed by $\mathcal{K}$ in a free message on k, therefore we have to use the context

$$\mathbf{V}_k = (\nu k)([\,] \mid k \Leftarrow (\lambda x)\mathbf{I})$$

(formally one should also restrict the free names of P).

LEMMA (COMPUTATIONAL ADEQUACY) 6.13. $k \notin \mathsf{fn}(P) \;\Rightarrow\; P{\Downarrow} \;\Leftrightarrow\; \mathbf{V}_k[\mathcal{K}(P,k)]{\Downarrow}$

The Theorem is an easy consequence of this lemma. The CPS transform is not fully abstract, that is, the converse of the Theorem does not hold. The situation is similar to the one of λ encoded into π (see [8,9]), namely η-expansion is not preserved for instance: we have seen that $x \sqsubseteq (\lambda y)xy$ but the encodings applied to k are distinguished, by means of the context $\mathbf{C} = (\nu k)(\nu x)([\,] \mid x \Leftarrow (\lambda z)\mathbf{I})$.

7. Conclusion

We have presented the blue calculus, which could serve as a basis for the design of a programming language combining functional (i.e. higher-order) and concurrent features. The calculus smoothly integrates both the λ-calculus and the π-calculus, together with their typing and sorting systems. We think that the inclusion $\pi \subset \pi^*$ sheds new light on the meaning of the π-calculus primitives. It has occurred to several researchers that a message is an application, and that the name of an input prefix may be regarded as the location of a resource, rather than a channel, but this intuitive interpretation was not formalized. Similarly, it was figured by some people that the π-calculus encodings of evaluation strategies in the λ-calculus correspond to CPS transforms, but again these remained intuitive analogies.

We think that the relationships between the two calculi are more clearly demonstrated in the unifying framework of the blue calculus. The understanding we get of the π-calculus from within π^* may be summarized as follows: a main theme of the "process algebra" approach to concurrency was, quoting Milner [18], that "*if naming is involved in communicating* (...) *and in locating and modifying data, then we look for a way of treating data-access and communication as the same thing*". This led him, as early as in CCS, to "*viewing data as a special kind of process.*" In π^* we rather took the view that communication is a special kind of data-access – though obviously they are both formally the same.

The π^*-calculus and its type system is clearly, as we presented it in this paper, at an early stage of existence, and we are further developing it, working on more refined type systems and syntax extension to deal with "higher-order concurrent object programming". To conclude this work, let us just discuss a little the syntax. We think that the restricted syntax introduced in the last section, based on a distinction between variables and references, is worth promoting as the right one. For instance, it allows us to define a notion of *closed term* which is stable by reduction, namely P is closed if it contains no free variable and no free declared reference, i.e. $\mathsf{decl}(P) = \emptyset$ – it may still contain free names, that refer to resources which are not yet defined.

The restriction that references should not be abstracted seems also useful if we want to extend the calculus to deal with *records*. To this end we could add the *matching* construct $[u = v]P$ (see [16]). Then, since a record R is just a mapping from a set of names $\{\ell_1, \ldots \ell_n\}$ to values, it can obviously be represented in the calculus as follows – assuming that the values themselves are encoded into π^*:

$$R = (\lambda x)([x = \ell_1]R_1 \mid \cdots \mid [x = \ell_n]R_n) \qquad (x \text{ fresh})$$

so that field selection is just the application $(R\,\ell)$. Then we would expect that

$$R\,\ell_i \simeq R_i$$

However, in general these two terms can be separated by first putting them into $((\lambda \ell_i)[\,])\ell_j$. Clearly, we need to ensure that the field names of a record are constants, and a way to do this is to regard these names as references, and to forbid abstracting them.

To deal with "objects" we also need to represent "extensible" records. We could then add the mismatching construct $[u \neq v]P$, or a form of conditional branching, which is operationally equivalent to $([u = v]P \mid [u \neq v]Q)$ – though for typing purposes it would be better to introduce a new construct, denoted $\{\![u = v]P, Q\}$ for instance, whose operational meaning is given by the rules

$$\{\![u = u]P, Q\} \to P$$
$$\{\![r = s]P, Q\} \to Q \qquad r, s \in \mathcal{R},\ r \neq s$$

Let us just give an example showing how we can model objects with changing state. Suppose we want to define a "cell" object, that accepts methods read, for accessing the current value, and write, updating the content of the cell. The specification requires that performing a write operation, we get the *same* cell with a new value, or a reincarnation of it, not an updated copy. Then the cell must be a recursive object, though not duplicated upon invocation of the write operation. Let C be the following declaration, where we use $[u = v]P$ as an abbreviation for $\{\![u = v]P, (\nu u)\langle u \Leftarrow \mathsf{0}\rangle\}$:

$$C =_{\text{def}} \mathsf{cell} \Leftarrow (\lambda m)\{\![m = \mathsf{write}]c\,,\ [m = \mathsf{read}](v \mid cv)\}$$

Then we may let

$$\mathsf{Cell} =_{\text{def}} \mathsf{rec}\, c.(\lambda v)C =_{\text{def}} (\nu c)(c \mid \langle c = (\lambda v)C\rangle) \qquad (*)$$

so that $\mathsf{Cell}\,u$ evaluates in a deterministic way:

$$\mathsf{Cell}\,u \stackrel{*}{\to} (\nu c)(\langle c = (\lambda v)C\rangle \mid [u/v]C)$$

It can be checked that these two terms are in fact semantically the same, and therefore we have

$$(\mathsf{Cell}\ u \mid (\mathsf{cell\ write})w) \stackrel{*}{\to}\simeq \mathsf{Cell}\ w$$
$$(\mathsf{Cell}\ u \mid (\mathsf{cell\ read})v_1 \cdots v_k) \stackrel{*}{\to}\simeq (\mathsf{Cell}\ u \mid uv_1 \cdots v_k)$$

Now suppose that we want to have a "global" declaration of a cell object, that we could instantiate into a local storage object s. The easiest way to perform the renaming of s into cell would be to abstract cell from Cell, but this violates our convention about declared names. Another way, frequently used in the π-calculus to encode values, is to pass cell around as a private name, argument of a message "make_cell" (denoted n below). In the same message, one may pass a private name init on which the local Cell will receive an initial value. Then we write the cell object as follows:

$$(\mathsf{cell\_obj} = (\lambda n)(\nu\mathsf{cell})(\nu\mathsf{init})(n\ \mathsf{cell\ init} \mid \mathsf{init} \Leftarrow \mathsf{Cell}))$$

(here Cell is not a reference, but a meta-variable that stands for the definition (*) above, otherwise (νcell) would have no effect). A local copy of this object for a process P, with initial value v, is created as follows:

$$(\nu s)\big(P \mid (\nu n)(\mathsf{cell\_obj}\ n \mid n \Leftarrow (\lambda pr)(\langle s = p\rangle \mid rv))\big)$$

This style of programming, although very close to the one in force in the π-calculus, exploits the higher-order character of π^*: Cell is a higher-order process, and it may contain higher-order values, as shown by the use of the read operation.

REFERENCES

[1] S. ABRAMSKY, C.-H.L. ONG, *Full abstraction in the lazy lambda-calculus*, Information and Computation 105 (1993) 159-267.

[2] R. AMADIO, L. LETH, B. THOMSEN, *From a concurrent λ-calculus to the π-calculus*, FCT'95, Lecture Notes in Comput. Sci. 965 (1995) 106-115.

[3] A. APPEL, *Compiling with Continuations*, Cambridge University Press (1992).

[4] G. BERRY, G. BOUDOL, *The chemical abstract machine*, Theoretical Comput. Sci. 96 (1992) 217-248.

[5] G. BOUDOL, *Asynchrony and the π-calculus*, INRIA Res. Report 1702 (1992).

[6] G. BOUDOL, *The λ-calculus with multiplicities*, INRIA Res. Report 2025 (1993).

[7] G. BOUDOL, *Lambda-calculi for (strict) parallel functions*, Information and Computation 108 (1994) 51-127.

[8] G. BOUDOL, C. LANEVE, *λ-Calculus, multiplicities and the π-calculus*, INRIA Res. Report 2581 (1995).

[9] G. BOUDOL, C. LANEVE, *The discriminating power of multiplicities in the λ-calculus*, Information and Computation 126 (1996) 83-102.

[10] M. BOREALE, *On the expressiveness of internal mobility in name-passing calculi*, CONCUR'96, Lecture Notes in Comput. Sci. 1119 (1996) 163-178.

[11] C. FOURNET, G. GONTHIER, *The reflexive CHAM, and the join calculus*, POPL'96 (1996) 372-385.

[12] A. GIACALONE, P. MISHRA, S. PRASAD, FACILE: *a symmetric integration of concurrent and functional programming*, TAPSOFT'89, Lecture Notes in Comput. Sci. 352 (1989) 184-209.

[13] R. HARPER, M. LILLIBRIDGE, *Polymorphic type assignment and CPS conversion*, LISP and Symbolic Computation 6 (1993) 361-380.

[14] J. LAUNCHBURY, *A natural semantics for lazy evaluation*, POPL'93 (1993) 144-154.

[15] A.R. MEYER, M. WAND, *Continuation semantics in typed lambda-calculi*, Lecture Notes in Comput. Sci. 193 (1985) 219-224.

[16] R. MILNER, J. PARROW, D. WALKER, *A calculus of mobile processes*, Information and Computation 100 (1992) 1-77.

[17] R. MILNER, *Functions as processes*, Math. Struct. in Comp. Science 2 (1992) 119-141. Preliminary version in INRIA Res. Report 1154.

[18] R. MILNER, *The polyadic π-calculus: a tutorial*, Technical Report ECS-LFCS-91-180, Edinburgh University (1991) Reprinted in *Logic and Algebra of Specification*, F. Bauer, W. Brauer and H. Schwichtenberg, Eds, Springer Verlag, 1993, 203-246.

[19] C. MURTHY, *A computational analysis of Girard's translation and LC*, LICS'92 (1992) 90-101.

[20] B. PIERCE, *Programming in the π-calculus – An experiment in concurrent language design*, available electronically, Computer Lab. Cambridge (1995).

[21] G. PLOTKIN, *Call-by-name, call-by-value and the λ-calculus*, Theoret. Comput. Sci. 1 (1975) 125-159.

[22] J.H. REPPY, CML: *a higher-order concurrent language*, ACM SIGPLAN'91 PLDI Conference, SIGPLAN Notices 26 (1991) 293-305.

[23] D. SANGIORGI, *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*, PhD Thesis, Department of Computer Science, The University of Edinburgh (1993).

[24] D. SANGIORGI, *From π-calculus to higher-order π-calculus – and back*, TAPSOFT'93, Lecture Notes in Comput. Sci. 668 (1993) 151-166.

[25] D. SANGIORGI, *The lazy lambda-calculus in a concurrency scenario*, Information and Computation 120 (1994) 120-153.

[26] G. SMOLKA, *A foundation for higher-order concurrent constraint programming*, in Constraints in Computational Logics, Lecture Notes in Comput. Sci. 845 (1994) 50-72.

[27] C. TALCOTT, Ed., *Special Issue on Continuations*, LISP and Symbolic Computation 6 & 7 (1993).

[28] D. TURNER, *The Polymorphic Pi-calculus: Theory and Implementation*, Ph.D. Thesis, University of Edinburgh (1995).

[29] V. VASCONCELOS, K. HONDA, *Principal typing schemes in a polyadic π-calculus*, CONCUR'93, Lecture Notes in Comput. Sci. 715 (1993) 524-538.