An investigation on the optimal implementation of processes*

Cosimo Laneve

Dipartimento di Scienze dell'Informazione, Bologna, Italy

Abstract

We undertake the analysis of Lévy's notion of family relation in a concurrent scenario by providing an optimal implementation of an higher-order, concurrent language. The language considered turns out to be a sub-language of (pure) *Concurrent ML*, where transmissible processes cannot show up any nondeterminism.

1 Introduction

In the last years a new class of languages is gaining prominence: the so-called *higher order concurrent languages*. Among them we recall *Concurrent ML* [13], *Facile* [4] and π-calculus [10]. These languages bring together two fundamental computational paradigms of computer science, functional programming and concurrency, with the aim of developing a support for applications requiring a combination of complex computations and concurrent/distributed computing. For instance, a manifest feature of higher order concurrent languages is the possibility to send complex data, such as processes and functions. This feature allows to describe in a direct way those facilities of distributed programming as code transmission.

One open problem in this area is how much the functional model of computation can be adapted for modeling (higher order) concurrent features. This problem is analyzed here through the looking glasses of the implementations. In particular we consider the algorithms already defined for implementing (pure) functional languages and study their adaptations in order to encompass concurrent facilities. The algorithms we study are the so-called *optimal implementations* [7, 5, 2], which are graph rewriting systems that have been provided for implementing the λ-calculus.

*Partially Supported by the ESPRIT Basic Research Project 6454 - CONFER

1.1 The notion of optimality in a concurrent scenario

The optimal implementation (in the sense of Lévy [9]) of a functional language relies on the principle that no work (i.e. no reduction) is ever duplicated. In particular two copies of the same expression are kept shared as much as possible in order to be evaluated at most once. Up to our knowledge, investigations about optimality have been never attempted in concurrency theory. This means that, foremost, we must clarify when an implementation of a concurrent language may be called "optimal". To this aim, take the following prototype concurrent language:

$$p \ ::= \ 0 \ \mid \ x \ \mid \ \gamma x.\,p \ \mid \ \overline{\gamma}\,p.\,p \ \mid \ p\,|\,p$$

where the *parallel composition* "$|$" is monoidal with 0 as identity; the *receiver* $\gamma x.\,p$ binds the *free occurrences* of x in p and the *sender* $\overline{\gamma}\,p.\,q$ transmits the process p along the channel γ. The definitions of free variables and substitution are the standard ones (see [3]).

The computational engine is provided by the *communication rule* below:

$$\gamma x.\,p\,|\,\overline{\gamma}\,r.\,q \ \rightarrow \ p[{}^{r}/_{x}]\,|\,q$$

where $p[{}^{r}/_{x}]$ denotes the substitution of r for the free occurrences of x in p.

Observe that, in the above language, the communication rule is the unique mechanism for performing work. Therefore in this context "optimality" means "avoiding duplication of the same communication". For instance, consider the process

$$\underbrace{\gamma\, x.\,(x\,|\,x)\,|\,\overline{\gamma}}_{B}(\,\overbrace{\lambda y.\,y\,|\,\overline{\lambda}}^{A}0.\,0\,).0$$

and perform the communication B. We obtain the process

$$\overbrace{\lambda y.\,y\,|\,\overline{\lambda}}^{A}0.\,0 \,\mid\, \overbrace{\lambda y.\,y\,|\,\overline{\lambda}}^{A}0.\,0$$

where the communication A has been duplicated. However the two communications A must be considered as two instances of the same *internal computation* and there is no sensible reason to perform it twice. An "optimal" implementation of the concurrent language should avoid such duplication. In functional languages this is usually achieved by means of suitable mechanisms keeping shared the expression $\lambda y.\,y\,|\,\overline{\lambda}0.\,0$.

1.2 The nondeterminism is troublesome

Let's go further on. Consider the process

$$\underbrace{\gamma\, x.\,(x \mid x) \mid \overline{\gamma}}_{B}\,(\,\overbrace{\lambda y.\, y \mid \overline{\lambda}}^{A}\, 0.\, 0\,)\,.0 \mid \overline{\lambda} p.\, q$$

and perform again the communication B. We obtain the process

$$\underbrace{\overbrace{\lambda y.\, y \mid \overline{\lambda}}^{A}\, 0.0 \mid \underbrace{\overbrace{\lambda y.\, y \mid \overline{\lambda}}^{A} 0.0 \mid \overline{\lambda}}_{C} p.\, q}_{D}$$

where one of the two communications A may be "broken" by the attempt of establishing a communication involving the process $\overline{\lambda} p.\, q$. That is, the same subprocess, in this case $\lambda y.\, y$, may be involved in different communications. Clearly, the evolution of a process depends, in general, on the choice performed at each step. This phenomenon, which misses in functional languages, is quite usual in concurrency theory and it is called (internal) *nondeterminism.*

The constraint that an optimal implementation must never duplicate work has subtle consequences when nondeterminism shows up. In particular, sharing communications at a given step of the computation means "centralizing the choices concerning them" (that is, performing all the choices at the same time). In the example above we have two possibilities: (a) recognizing that the communications A are copies and then firing them at once or (b) firing one of the copies of A and then performing C or D. Choose the possibility (b). Once having performed one copy of A, we are obliged to fire C or D otherwise we should invalidate the requirement of optimality. This means that the second copy of A must be "disallowed" in favour of C or D. However the redexes C and D may be "virtual", in the sense that they will become redexes in the future (because, for instance, the process $\overline{\lambda} p.\, q$ is inside the scope of a send or a receive). To this aim we must keep track of what communications have been performed in the past and forbid the contraction of copies. Pursuing this approach on, we have to admit processes which must be considered in normal form even if they have subterms which are redexes (because these redexes have been disallowed).

Keeping track of the communications which have been performed along a computation is surely expensive. This may be avoided by deciding that all the copies of a communication must be always kept shared, namely their evaluation must be performed at the same time (the computation is *complete*). This choice is surely debatable, since interesting computations may be left out. Take for instance the process

$$\overbrace{\lambda y.\, 0 \mid \overline{\lambda}}^{A}\, 0.\, 0 \mid \overbrace{\lambda y.\, y \mid \overline{\lambda}}^{A} 0.\, 0 \mid \lambda x.\, (x \mid \overline{\lambda} x.\, 0) \mid \overline{\lambda}\, (\lambda x.\, (x \mid \overline{\lambda} x.\, 0)).\, 0$$

This process never terminates if the two copies of A are kept shared. On the other hand, if the sharing is broken, the process evaluates to 0.

1.3 Avoiding troubles

At the present stage of our research we want to avoid any decision limiting the nondeterminism that a process may show up. Therefore we have decided to consider a syntactical restriction of the language described in Subsection 1.1, where nondeterminism cannot be shown by processes that are copied. This is obtained by constraining those processes that can be sent as messages to be synchronous (the possible parallelism forces the communications between sub-processes through an operation which resembles very much to function application). The more general operation of parallel composition is left for other processes.

Surely our decision is too drastic: the syntactical restrictions on the calculus make it scanty and may be weakened. This does not scare us too much, since the aims of this paper are:

1. to provide a direct extension of the standard optimal λ-calculus implementations to a (simple) concurrent language. In particular, we aim at enriching the BOHM evaluator [1] with concurrent primitives, in order to define an optimal implementation of a fragment of CML.
2. to start a deeper analysis of what means (and/or what implies) to not duplicate processes.

In this respect, we have chosen a language that fit better with optimal implementations, leaving the problem of generalizing it to future research.

1.4 Outline of the paper

The outline of the paper is as follows. In Section 2 we present our higher order concurrent language and its communication mechanism. In the same section we also define a labeling which allows to formalize the notion of family, i.e. those communications which are copies, and of optimal implementation, i.e. implementations where families have unique representations. The implementation is described in Section 3 and its correctness and optimality is stated in Section 4 (the proofs are provided in the full paper). In Section 5 we discuss some future themes of research.

2 The calculus

Consider the set of terms generated by the following grammar:

$$\begin{array}{rcl} t & ::= & 0 \mid x \mid \gamma x.t \mid \overline{\gamma} t.t \mid t \| t \\ p & ::= & t \mid \gamma x.p \mid \overline{\gamma} t.p \mid p \| p \mid p | \cdots | p \end{array}$$

where $\boldsymbol{x}$ stays for a generic variable, γ is a port into a set Γ. The expressions generated by the non terminal $\boldsymbol{t}$ are called *synchronous processes* and are ranged over by $\boldsymbol{t}, \boldsymbol{s}, \cdots$; the expressions generated by the non terminal $\boldsymbol{p}$ are called *processes* and ranged over by $\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \cdots$. The process 0 is the terminated process; $\gamma\boldsymbol{x}.\,\boldsymbol{p}$ is the process which waits for a message $\boldsymbol{t}$ on the channel γ and then behaves as $\boldsymbol{p}[^{t}/_{x}]$; $\overline{\gamma}\boldsymbol{t}.\,\boldsymbol{p}$ is the process that sends a message $\boldsymbol{t}$ on the channel γ and then behaves as $\boldsymbol{p}$; $\boldsymbol{p}_1 \,|\cdots|\, \boldsymbol{p}_k$ is the parallel composition of the processes $\boldsymbol{p}_1, \cdots, \boldsymbol{p}_k$: their communication is not forced, in the sense that they may also interact with processes in the environment; $\boldsymbol{p} \,||\, \boldsymbol{q}$ is the synchronous parallel composition of the $\boldsymbol{p}$ and $\boldsymbol{q}$: they must always communicate. Notice that there is a family of parallel operators "$|$", every one with a different ariety. For simplicity these operators are noticed in the same way.

Processes are taken up-to the congruence $\equiv$ generated by the following axioms:

- $\boldsymbol{p} = \boldsymbol{q}$, if $\boldsymbol{p}$ and $\boldsymbol{q}$ are $\boldsymbol{\alpha}$-equivalent;
- $\boldsymbol{p} \,|\, 0 = \boldsymbol{p}$, $\boldsymbol{p} \,|\, \boldsymbol{q} \;=\; \boldsymbol{q} \,|\, \boldsymbol{p}$, $\boldsymbol{p} \,|\, (\boldsymbol{p}_1 \,|\cdots|\, \boldsymbol{p}_k) \;=\; \boldsymbol{p} \,|\, \boldsymbol{p}_1 \,|\cdots|\, \boldsymbol{p}_k$;
- $\boldsymbol{p} \,||\, 0 = \boldsymbol{p}$, $\boldsymbol{p} \,||\, \boldsymbol{q} = \boldsymbol{q} \,||\, \boldsymbol{p}$.

Remark 2.1 *Observe that the* synchronous composition $||$ *is a binary, non associative operation, which has 0 as identity. There are many analogies between* $||$ *and the usual function application. However, the main differences between these two operators are:*

1. $||$ *may deadlock. This is the case in the processes* $\gamma\boldsymbol{x}.\,\boldsymbol{p} \,||\, \overline{\mu}\boldsymbol{t}.\,\boldsymbol{q}$ $(\gamma \neq \mu)$, $\gamma\boldsymbol{x}.\,\boldsymbol{p} \,||\, \mu\boldsymbol{y}.\,\boldsymbol{q}$ *or* $\overline{\gamma}\boldsymbol{s}.\,\boldsymbol{p} \,||\, \overline{\mu}\boldsymbol{t}.\,\boldsymbol{q}$.
2. *there is no ordering on the arguments of* $||$.

In the following we shall use the definitions of substitution and free variables of a term $\boldsymbol{M}$ (notation $\mathsf{fv}(\boldsymbol{M})$). These are the standard ones (see [3]).

The evaluation mechanism is provided by the following two rewriting rules:

$$\gamma\boldsymbol{x}.\,\boldsymbol{t} \,||\, \overline{\gamma}\,\boldsymbol{t}'.\,\boldsymbol{s} \;\rightarrow\; \boldsymbol{t}[^{t'}/_{x}] \,||\, \boldsymbol{s}$$

$$\gamma\boldsymbol{x}.\,\boldsymbol{p} \,|\, \overline{\gamma}\,\boldsymbol{t}.\,\boldsymbol{q} \;\rightarrow\; \boldsymbol{p}[^{t}/_{x}] \,|\, \boldsymbol{q}$$

where the notation $\boldsymbol{p}[^{t}/_{x}]$ denotes the substitution of $\boldsymbol{t}$ for the free occurrences of $\boldsymbol{x}$ in $\boldsymbol{p}$.

Example 2.2 *We may easily encode* λ*-calculus in the above language. It is enough to use synchronous par instead of application. The encoding* $[\![\;]\!]$ *is defined by:*

$$\begin{array}{rcl} [\![\boldsymbol{x}]\!] & = & \boldsymbol{x} \\ [\![\lambda\boldsymbol{x}.\,\boldsymbol{M}]\!] & = & \lambda\boldsymbol{x}.\,[\![\boldsymbol{M}]\!] \\ [\![\boldsymbol{M}\boldsymbol{N}]\!] & = & [\![\boldsymbol{M}]\!] \,||\, \overline{\lambda}[\![\boldsymbol{N}]\!].\,0 \end{array}$$

Observe that a non commutative operator as function application is modeled through a commutative one (the $\|$). This is not problematic, since one of the two argument of the $\|$ is a sender and the other is a (possible) receiver.

Remark 2.3 *By the above example, it is clear that we may implement the following sub-language of Concurrent ML [13]:*

$$e \;::=\; x \mid \texttt{fun}\, x \Rightarrow e \mid e\, e \mid \texttt{let}\, x = e\, \texttt{in}\, e \mid \texttt{let rec}\, x = e\, \mathit{in}\, e$$

$$\begin{array}{lll} p & ::= & e \mid (\,) \mid \texttt{fun}\, x \Rightarrow p \mid \texttt{let}\, x = p\, \texttt{in}\, p \mid \texttt{channel}() \mid \texttt{send}(c, e); p \\ & & \texttt{accept}(c); p \mid \texttt{spawn}(p); \cdots; \texttt{spawn}(p) \end{array}$$

Of course we may encode larger parts of Concurrent ML, since our transmissible processes may deadlock. However the above language should give a strong flavour of the expressive power of our calculus.

Definition 2.4 *Let p be a process. A* redex *in p is a subterm of shape $\gamma x.\, p \mid \overline{\gamma}\, t.\, q$ or $\gamma x.\, p \;\|\; \overline{\gamma}\, t.\, q$. We assume that redexes are uniquely identified by names u, v, w.*

2.1 The labeling

Let A be a countable set of *atomic labels.* The labels we shall consider are terms generated by the following grammar:

$$\ell \;::=\; A \mid \ell \cdot \ell' \mid \overline{\ell} \mid \underline{\ell} \mid \boxed{\ell}$$

The labeled processes are the terms e generated by the grammar:

$$\begin{array}{lll} \mathsf{c} & ::= & \ell(0) \mid \ell(x) \mid \ell(\gamma x.\, \mathsf{c}) \mid \ell(\overline{\gamma}\, \mathsf{c}.\, \mathsf{c}) \mid \ell(\mathsf{c}) \| \ell(\mathsf{c}) \\ \mathsf{e} & ::= & \mathsf{c} \mid \ell(\gamma x.\, \mathsf{e}) \mid \ell(\overline{\gamma}\, \mathsf{c}.\, \mathsf{e}) \mid \ell(\mathsf{e}) \,\|\, \ell(\mathsf{e}) \mid \ell(\mathsf{e}) \mid \cdots \mid \ell(\mathsf{e}) \end{array}$$

Labeled processes are quotiented by the same axioms of the structural equivalence as for processes plus the axioms

$$\mathsf{e} \,\|\, \ell(0) = \mathsf{e} \qquad \mathsf{e} \mid \ell(0) = \mathsf{e} \qquad \ell(\mathsf{e} \mid \cdots \mid \mathsf{e}') = \ell(\mathsf{e}) \mid \cdots \mid \ell(\mathsf{e}') \qquad \ell(\ell'(\mathsf{e})) = \ell \cdot \ell'(\mathsf{e})$$

The rewriting rules of the labeled processes are:

$$\ell(\gamma x.\, \mathsf{e}) \,\|\, \ell'(\overline{\gamma}\, \mathsf{e}'.\, \mathsf{e}'') \;\rightarrow\; \overline{\ell \cdot \ell'}(\mathsf{e}[\underline{\ell \cdot \ell'}(\mathsf{e}')/x] \,\|\, \boxed{\ell \cdot \ell'}(\mathsf{e}''))$$

$$\ell(\gamma x.\, \mathsf{e}) \mid \ell'(\overline{\gamma}\, \mathsf{e}'.\, \mathsf{e}'') \;\rightarrow\; \overline{\ell \cdot \ell'}(\mathsf{e}[\underline{\ell \cdot \ell'}(\mathsf{e}')/x] \mid \boxed{\ell \cdot \ell'}(\mathsf{e}''))$$

where ℓ and ℓ' have no common prefix, namely there is no $\ell^\flat$ such that $\ell = \ell^\flat \cdot \ell_1$ and $\ell' = \ell^\flat \cdot \ell_1'$. The set $\{\ell, \ell'\}$ will be called *label of interaction.*

The rewriting rule involving par-composition deserves some comments. Due to the axiom $\ell(\mathsf{e} \mid \mathsf{e}') = \ell(\mathsf{e}) \mid \ell(\mathsf{e}')$, we have, for instance,

$$\ell(\ell'(\gamma x.\, p) \mid \ell''(\overline{\gamma} t.\, q)) = \ell \cdot \ell'(\gamma x.\, p) \mid \ell \cdot \ell''(\overline{\gamma} t.\, q)$$

where we assume that ℓ' and ℓ'' have no common prefix. But $\ell(\ell'(\gamma x.\, p) \mid \ell''(\overline{\gamma} t.\, q)) \rightarrow \ell(\overline{\ell' \cdot \ell''}(p[\underline{\ell' \cdot \ell''}(t)/x] \mid \boxed{\ell' \cdot \ell''}(q)))$ and, in order to be consistent with the axiomatization, we must also guarantee that

$$\ell \cdot \ell'(\gamma x.\, p) \mid \ell \cdot \ell''(\overline{\gamma} t.\, q) \rightarrow \ell(\overline{\ell' \cdot \ell''}(p[\underline{\ell' \cdot \ell''}(t)/x] \mid \boxed{\ell' \cdot \ell''}(q)))\ .$$

This is the reason for keeping a suffix of the labels of the sender and of the receiver as label of interaction of a par-composition. Remark that this operation must be safe w.r.t. the problem of optimal reductions, because labeling is used in order to formalize the notion of optimal sharing (see Definition 2.6 below). Actually this is the case, since there is never sharing on redexes composed with the par operator (see Proposition 2.8).

Clearly there is a strong relationship between a labeled process and the one obtained by erasing labels. In particular, they perform the same reductions. We will not formalize this correspondence, since it is quite obvious, and in the following we will always go from labeled processes to the (corresponding) unlabeled ones (and vice versa).

Definition 2.5 *Let* e *be a labeled process where every label is atomic and different from any other (this property is described by the proposition* init*). Let* $\mathsf{e} \rightarrow^* \mathsf{e}'$*. Two redexes in* e' *are in the same* family *if and only if they have the same label of interaction.*

Redexes in the same family must be considered as *copies*, that is they are obtained by duplications performed into the past. Remark that families in the initial term have always cardinality 1 (because there is no past).

Definition 2.6 *An implementation is* optimal *if the representation of redexes in the same family is kept unique.*

Said with other words, an optimal implementation performs no useless duplication.

Remark 2.7 *The syntactical counterpart of a reduction of an optimal implementation is the contraction of a* complete family *of redexes, namely all the redexes that have the same label of interaction. These contractions may reduce the possible nondeterminism because the choices concerning the redexes in a family must be done once, for every redex (see also the Introduction). For instance, take the process*

$$r \;=\; \underline{\gamma x.\, \gamma y.\, p} \mid \underline{\underline{\overline{\gamma} t.\, q}} \mid \underline{\gamma x.\, \gamma y.\, p} \mid \underline{\underline{\overline{\gamma} t.\, q}}$$

where processes with the same underlining must be considered as copies. Then we may reduce r as follows (assuming $x \neq y$ and $y \notin \mathsf{fv}(t)$):

$$r \to \gamma y.\, p[{}^{t}/_{x}] \,|\, \gamma x.\, \gamma y.\, p \,|\, \overline{\gamma}\, t.\, q \,|\, q \;=\; r'$$

There are two redexes in r': one between the process $\gamma y.\, p[{}^{t}/_{x}]$ and $\overline{\gamma}\, t.\, q$ and the other, which is in the same family of the first reduction, between $\gamma x.\, \gamma y.\, p$ and $\overline{\gamma}\, t.\, q$. If we choose the complete family of the first redex to be fired as first reduction then we loose at the second step the redex between $\gamma y.\, p[{}^{t}/_{x}]$ and $\overline{\gamma}\, t.\, q$.

Proposition 2.8 *Let $p \to^* q$ and $\gamma x.\, q' \,|\, \overline{\gamma}\, r.\, r'$ be a redex occurring in q. The cardinality of the family of this redex is always 1.*

That is, redexes built with the parallel composition have never copies. This has a strong consequence on the implementation, since only synchronous processes may appear in the scope of a duplicator.

3 The implementation of processes

Let us come to the optimal implementation of our processes. As stated in Definition 2.6, the aim is to avoid duplications of redexes which are in the same family. This is performed by means of a graphical representation of expressions, where sharing is performed by ad-hoc nodes (for this reason the graphs will be called *sharing graphs*). That is, besides the nodes needed for describing the operators of our calculus, called *proper nodes*, we need a set of nodes for implementing sharing, (better, *optimal sharing*, in the sense of Definition 2.6) as for usual λ-calculus (see [5]). These latter nodes are called *control nodes*. The set of nodes of the evaluator is illustrated in Figure 1. Control nodes (erasing, croissant, bracket and fan) interact

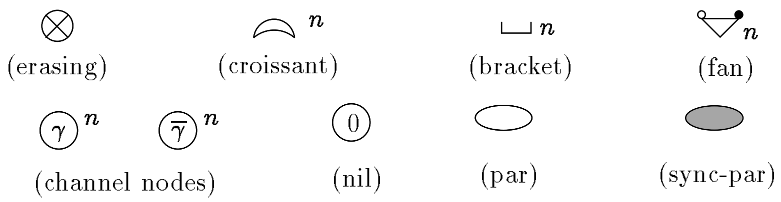


Figure 1: The set of nodes of the evaluator

with other nodes through the port in the bottom, which is called *principal port*. The proper nodes are channel nodes, nil, par and sync-par. Channel nodes and nil interact with the other nodes through the port in the top. Par and sync-par may interact with every port. The port in the top of the proper nodes is called *output* *port*. The other ports are called *argument ports*, except for the port at the left, in the bottom of a channel node γ, which is called *bound port*.

Remark that the synchronous par has exactly two argument ports, whilst the par may have several arguments. This because it represents an unordered list of processes (due to the commutativity and associativity axioms).

There are several ways for explaining the purposes and the meaning of control nodes. For example, in [5] a linear logic metaphor is used. Alternatively, control nodes may be understood from the semantic point of view (i.e. the terms represented by sharing graphs) [7, 5, 2].

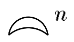
3.1 The encoding of processes

For each integer n and process p, we define a graph $\mathcal{P}_n(p)$ by induction on p together with its set of *free nodes* which represent the free variables of p. The translation of a process p is defined to be $\mathcal{P}(t) = \mathcal{P}_0(t)$.

Nil 0. The encoding of 0 is:

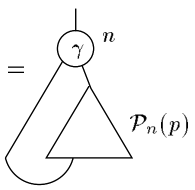
$\mathcal{P}_n(0) =$ 

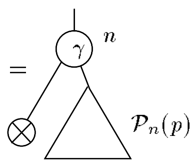
Variable x.

$\mathcal{P}_n(x) =$ 

The n-indexed croissant represents the (free) variable x.

Input $\gamma x.p$. There are two cases, according to the variable x does or does not appears in p. When x is in p we connect the the free node of p representing x with the bound port of γ (in this case the free nodes of $\gamma x.p$ decrease of one). Otherwise the bound port of γ is connected with an erasing node. The two cases are drawn below:

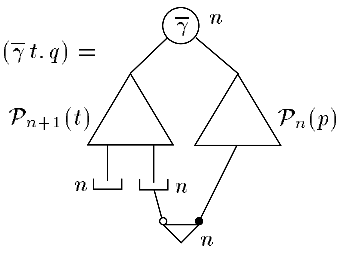
$\mathcal{P}_n(\gamma x.p) =$ 

$\mathcal{P}_n(\gamma x.p) =$ 

We warn the reader that erasing nodes produce garbage. As far as we know there is no set of rules able to eliminate all the garbage which may appear in sharing graphs. For this reason in our basic system we avoid to deal with garbage collection rules.

Output $\overline{\gamma}t.\,p$. The output of a term t over a channel γ and continuation p is encoded as follows:

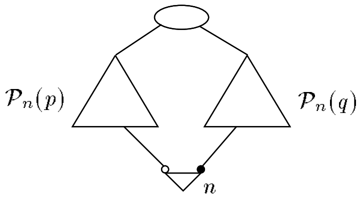
$\mathcal{P}_n(\overline{\gamma}\,t.\,q) =$



Remark that the encoding of t is at a deeper level then the other part of the graph (in the linear logic terminology, t is inside a *box* because it may be duplicated and/or erased [5]). The free nodes corresponding to the same variable in t and in p are linked together in $\mathcal{P}_n(\overline{\gamma}t.\,p)$ by means of a fan.

Parallel composition $p_1 \,|\, \cdots \,|\, p_k$. For simplicity, let us draw the case $p\,|\,q$. The general case $p_1 \,|\cdots|\, p_k$ is similar, putting a tree of fans in the bottom in order to implement the sharing of common variables.

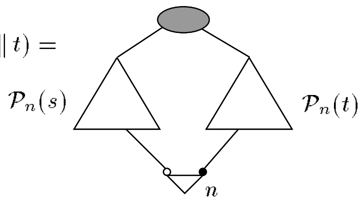
$\mathcal{P}_n(p\,|\,q) =$



As for the output, the free nodes corresponding to the same variable in p and in q are linked together in $\mathcal{P}_n(p\,|\,q)$ by means of a fan.

Synchronous parallel composition $s \parallel t$. The encoding of the synchronous parallel composition is quite similar to the one for the parallel composition:

$\mathcal{P}_n(s \parallel t) =$



The root of the term is represented by the node in the top of $\mathcal{P}(p)$. The free variables of p are represented by the free nodes of $\mathcal{P}(p)$.

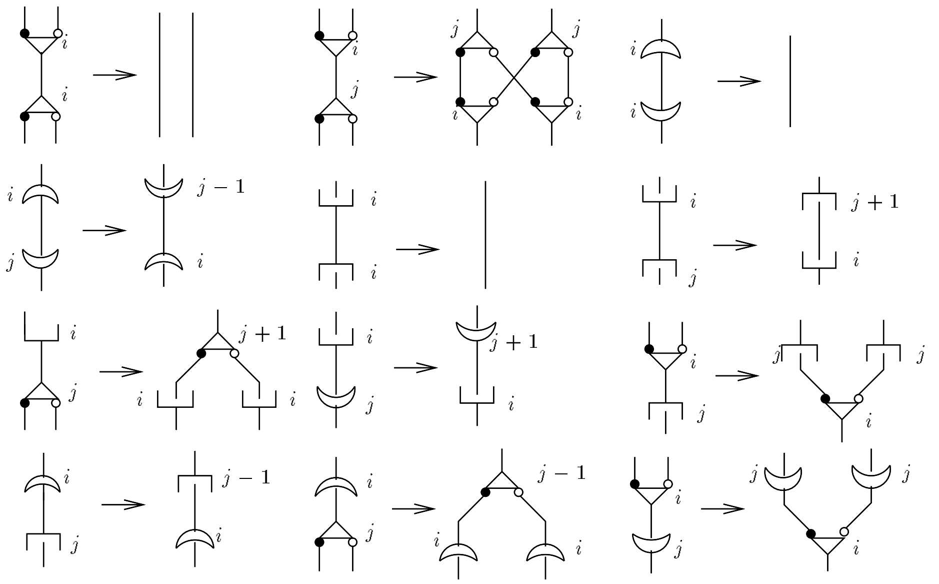


Figure 2: The control rules ($0 \leq i < j$)

3.2 The rewriting rules

Rewriting rules may be classified in four groups:

1. *control rules*, managing the interaction between control nodes;
2. *axiom rules*, implementing the axiomatization $\equiv$ of processes;
3. *communication rules*, performing the communication between two processes;
4. *interfacing rules*, describing the interaction between a control node and a proper node.

The discussion of these rules is in order.

1. Control Rules. These are the same 12 rules used for implementing optimal sharing in the λ-calculus (see [5], for instance). Control rules are illustrated in Figure 2.

2. Axiom Rules. These rules implement the axiomatization of processes. For instance we must take processes composed by the par node up-to commutativity, associativity and consider 0 as the identity. This is performed by the evaluator through a set of par-nodes taking a number n of arguments, for every $n > 0$. The rewriting rules are depicted in Figure 3. In the case of sync-par we have just one rule stating that $p \,||\, 0 = p$.

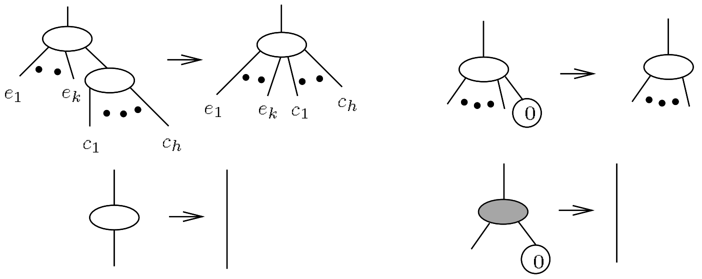


Figure 3: The axiom rules

3. Communication Rules. A communication between two processes, a sender and a receiver on the same channel, is performed when they are in a par context or in a sync-par context. In these cases we have a sort of generalization of β-reduction (where the second argument of the application has a continuation). The rules are

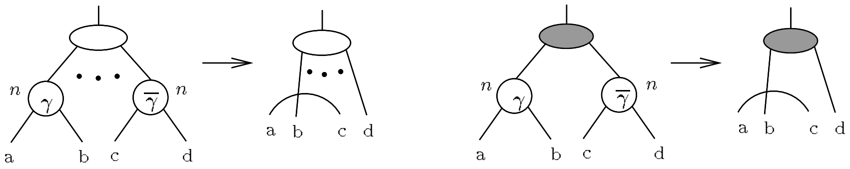


Figure 4: The communication rules

illustrated in Figure 4.

It is important to observe that the communication of processes connected by the par node may cause nondeterminism. A typical configuration is depicted in Figure 5, where a sender on the channel γ is in parallel with two receivers on the same channel. Then a choice must be done, favouring exactly one of the receivers.

4. Interfacing Rules. The interaction between control nodes and parallel facilities has been the main difficulty of this paper, since being too liberal with the transmissible processes is conflicting with optimality (see Section 1). For this reason we have constrained messages of our calculus to be synchronous processes. As a consequence, the evaluator holds the property that no control node is in front of one of the port of some par node. However, control nodes may be well in front of other proper nodes, different form the par. An example of process whose evaluation shows up an interaction between a fan and a sync-par is $\boldsymbol{\gamma x}.\, ((\boldsymbol{x}||\boldsymbol{N})\,|(\boldsymbol{x}||\boldsymbol{M}))\,|\,(\overline{\boldsymbol{\gamma}}\,(\boldsymbol{\gamma y}.\,(\boldsymbol{x}||\boldsymbol{P})).\,\boldsymbol{Q})$. The interactions between control nodes and proper nodes, called *interfacing rules*, are illustrated in Figure 6.

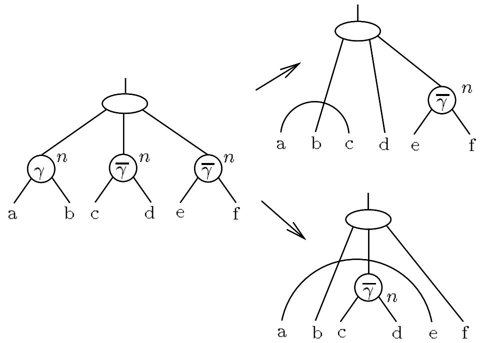


Figure 5: A case of nondeterminism

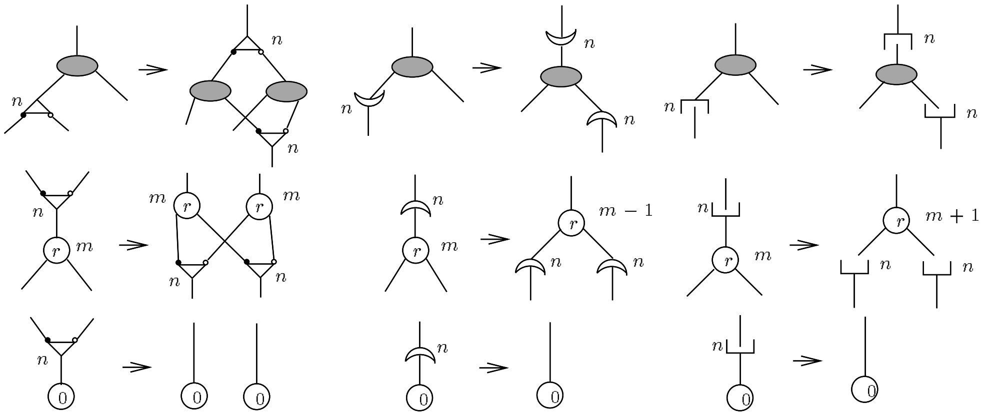


Figure 6: The interfacing rules ($n < m$, $r \in \{\gamma, \overline{\gamma}\}$)

Remark 3.1 *The optimal evaluator implements the computation process in terms of a set of local and microscopic moves. Observe that these elementary moves may run in parallel, which put forward a further issue: the possibility of implementing the parallelism of a process at a granularity which is different, better, finer than the one expressed by the process itself.*

Besides the critical pairs which cause nondeterminism (see Figure 5) there are other critical pairs, which concern par nodes, but are harmless.

Proposition 3.2 *Let* $\mathcal{P}(p) \to^* N$*,* $N \xrightarrow{u} N'$ *and* $N \xrightarrow{v} N''$*, such that* u *is an axiom rule and* v *is a communication rule. Then there exist* u'*,* v' *such that* $N' \xrightarrow{v'} N^+$ *and* $N'' \xrightarrow{u'} N^+$*.*

4 Correctness and optimality

The correctness of the implementation is based on a *read-back procedure* which gives the process represented by a sharing graph. The implementation is correct if, whenever N is a sharing graph obtained by evaluating $\mathcal{P}(p)$ and $N \to N'$, then the process read-back from N eventually reduces to the process read-back from N'. There is no space in this extended abstract for defining the read-back procedure and proving its correctness. We just outline the main ideas behind it and encourage the interested reader to read the full paper, which may be got by ftp [8].

The definition of the read-back procedure is provided by labeling edges of the sharing graphs through *contexts* and interpreting the nodes as *contexts transformers*. Expressions matching the sharing graphs are thus "unfoldings" of the graphs where only particular paths are considered (*consistent paths*), that is paths that behave well w.r.t. contexts.

Informally, the read-back behaves as follows. Initially, it is called with the root node $\mathtt{n}$ of the sharing graph (the outermost proper node). Then the read-back

1. creates a new node $\mathtt{m}$ of the same type as $\mathtt{n}$. The node $\mathtt{m}$ is the root of the syntax tree of the read-back term.

2. looks for all the arguments of $\mathtt{n}$. Namely the read-back takes all the consistent paths starting at the argument ports of $\mathtt{n}$, terminating at a proper node and never traversing proper nodes. Let $\{\varphi_1, \cdots, \varphi_k\}$ be such set of consistent paths. This means that the node $\mathtt{m}$ will have k arguments. Let us read-back the i-th argument.

3. let $\mathtt{n}_i$ be the node at the end of φ_i. There are two cases:

   (a) The proper node $\mathtt{n}_i$, possibly indexed by h, is accessed from the output port. This case is easy: we create a new node $\mathtt{m}_i$ corresponding to $\mathtt{n}_i$, connect the i-th argument port of $\mathtt{m}$ and the output port of $\mathtt{m}_i$. Then we reiterate the read-back starting from $\mathtt{n}_i$.

   (b) The node $\mathtt{n}_i$ is accessed from a bound port. Then the binder is one of the nodes got by reading-back $\mathtt{n}_i$. We must determine what ancestor node of $\mathtt{m}_i$ in the syntax tree is the right binder. The problem is that there could be a lot of them and determining the right one is far from evident (there are two properties of sharing graphs which guarantee the existence and unicity of the binder: the *transparency property* and the *separation property*).

Let's switch to the issue of optimality. We have to prove that the representation of redex families in the sharing graphs is always unique. Therefore, the amount of sharing performed by sharing graph is maximal, since work is never duplicated. To this aim Lamping shows that suitable paths in sharing graphs, called *prerequisite chains*, cannot be completely duplicated. This sufficies since β-redexes are a particular case of prerequisite chains. Our case is more difficult than λ-calculus because of the presence of nodes which haven't a unique principal port, such as the par and sync-par. Consequently the mechanism of prerequisite chains must be generalized to graphs.

Summing up the results in [8] we may state:

Theorem 4.1 *The implementation of Section 3 is correct and optimal.*

An immediate consequence of the above theorem follows by a close inspection of the encoding in Example 2.2:

Corollary 4.2 *The encoding of λ-calculus in Example 2.2 is correct and optimal.*

5 Conclusion and future work

In this paper we have provided an optimal implementation of an higher-order concurrent language. This language allows parallelism, nondeterminism and synchronous communication. The unique constraint is that transmissible processes must be deterministic. Our programme of research aims at

1. developing an optimal implementation for calculi admitting every process to be transmitted. Besides the problems we discussed in the Introduction, we must also solve the problem of keeping shared copies of redexes when a request for breaking one of them appears. A possible way-out is to use a general form of fan, where sequences of control operators are absorbed into one. Such a technique has been already proposed by Guerrini [6] and proved to be advantageous in terms of efficiency and of the read-back procedure.

2. studying architectures for concurrent/distributed implementations of our evaluators. As we have remarked (see Remark 3.1), optimal algorithms describe processes at a finer level of granularity, where concurrent redexes have not to synchronize and, therefore, have a propensity towards parallelism.

Besides the above two items, there is another issue which intrigues us. Milner has recently introduced graphical representations of his *action calculi* (*pi-nets* in [11], *action graphs* in [12]). There are a lot of analogies between our sharing graphs and Milner's graphs. One for all the notion of interaction, which underlies the two systems. And surely we will not be surprised if someone will present sharing graphs in terms of Milner's action structures. What we consider interesting is the understanding of the deep relationships, if any, between the two systems. Such as, whether the mechanism of interaction is more "primitive" in one system or in the other. Or the existence of a "minimal set" of nodes, possibly got from the two systems, through which express concurrency.

Acknowledgements Many thanks to Andrea Asperti for the discussions we had together. The anonymous referees have provided good hints for improving the paper.

References

[1] A. Asperti, C. Giovannetti, and A. Naletto. The Bologna Optimal Higher-order Machine. Technical Report UBLCS-95-9, Department of Computer Science, University of Bologna, March 1987.

[2] A. Asperti and C. Laneve. Interaction Systems II: the practice of optimal reductions. To appear on Theoretical Computer Science. Also available by anonymous ftp at `cma.cma.fr`, in the directory `pub/papers/cosimo`, May 1993.

[3] H.P. Barendregt. *The Lambda Calculus.* North-Holland, 1985.

[4] A. Giacalone, P. Mishra, and S. Prasad. Facile: a symmetric intefration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121 – 160, 1989.

[5] G. Gonthier, M. Abadi, and J.J. Lévy. Linear logic without boxes. In *Proceedings* 7^{th} *Annual Symposium on Logic in Computer Science*, 1992.

[6] S. Guerrini. A linear algebra approach to optimal λ-graph reduction. Technical Report TR-13/94, Dipartimento di Scienze dell'Informazione, Universitá di Pisa, November 1994.

[7] J. Lamping. An algorithm for optimal lambda calculus reductions. In *Proceedings* 17^{th} *ACM Symposium on Principles of Programmining Languages*, pages 16 – 30, 1990.

[8] C. Laneve. An investigation on the optimal implementation of processes. Draft. Available by anonymous ftp at `cma.cma.fr` as `pub/papers/cosimo/new.ps.gz`. May, 1995.

[9] J.J. Lévy. *Réductions correctes et optimales dans le lambda calcul.* PhD thesis, Université Paris VII, 1978.

[10] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119 – 141, 1992.

[11] R. Milner. Pi-nets: a graphical form of π-calculus. In *ESOP 94*, volume 788 of *Lecture Notes in Computer Science*, pages 26 – 42. Springer-Verlag, 1994.

[12] R. Milner. Calculi for Interaction. To appear in Acta Informatica, 1995.

[13] J. H. Reppy. CML: a higher-order concurrent language. *SIGPLAN Notices*, 26(6):294 – 305, 1991.