Università di Pisa

Dipartimento di Informatica

Technical Report : TR-96-30

A Rewriting Implementation of pi-calculus

Patrick Viry

March 26, 1996

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

A Rewriting Implementation of π-calculus

Patrick Viry

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56100 Pisa, Italy
Email: viry@di.unipi.it

Abstract. We introduce a rewriting implementation of the reduction relation of π-calculus and prove its correctness. The implementation is based on terms with De Bruijn indices and an explicit substitution operator. The resulting rewrite rules need to be applied modulo a large and complex equational theory, and are only of theoretical interest. Applying the coherence techniques introduced in a previous paper, we transform this specification into an equivalent one that only requires rewriting modulo associativity and commutativity. This latter rewrite system can then be straightforwardly encoded in currently available rewriting-based programming languages. Finally, we sketch a possible application of this implementation as the basis for adding input-output capabilities to such languages.

1 Introduction

Rewriting interpreters are very nice toys for the theoretically minded. You can feed them equations and rules, and they will show you the possible derivations, or compute normal forms, many variations of this game are possible. You can even argue that these toys are of great interest to the programming community, since the semantical counterpart of rewriting, namely rewriting logic [Mes92, MOM93], can be seen as a general multiparadigm programming model (or also as a general logical framework), in which many common computational models can be described in a simple and natural way, and moreover can be composed while preserving a sound semantical basis [MMO94].

With all the developments of the theory of rewriting in the last two decades, you can even claim that it should soon be possible to use rewriting-based systems for "real" programming in the "real" world, and exhibit a truckload of arguments showing the effectiveness of this approach and its advantages to classical techniques.

Yet, you would not be able right now to write the canonical first program "Hello world" : currently available rewriting interpreters lack input-output capabilities.

As a model of real I/O, we propose to use the π-calculus [MPW92a, MPW92b], a process algebra exhibiting mobility and in some sense value passing. In order to map π-calculus into rewriting logic, we equip it with an explicit substitution calculus similar to that of $\lambda\upsilon$ [LRD94], and show how to obtain a rewrite system modulo AC simulating exactly the reduction relation of π-calculus.

So what is new ? Any functional language on the market also has I/O capabilities. The main difference is that in our case the behaviour of I/O is totally and *explicitely* specified, in the *same* theoretical framework than for instance a "functional" or "equational" program computing the values sent on an output channel. This is not trivial on a semantical point of view : functional computation and I/O have two different and a priori incompatible interpretations, one in terms of equality and the other in terms of transitions between states.

This is to contrast to the situation in functional programming. There, for the same practical considerations, I/O capabilities have been added in an ad-hoc manner by the *implementors* of systems. But I/O does not fit in the functional model, because input and output are not functions. Extensions of the functional model have been devised, often based on the concept of monads (see e.g. [Gor94]), but make a non trivial addition to the original simple model. Since I/O is mainly left unspecified, it is difficult to understand and reason about a program, for instance assert properties about its communication behaviour or study its internal parallelism.

In the case of rewriting logic, there is no need to artificially map an I/O calculus to an unrelated and unadequate basic framework such as λ-calculus. Instead, both of these calculi can be mapped to rewriting logic in a natural way, because rewriting logic does not make any assumption about the computational model (everything must be explicited), and their interaction can be studied in a simple and unified framework, using for instance the methods proposed in [Vir95].

The paper is structured as follows. In a first part, we argue the choice of π-calculus as our basis for describing I/O, then give an executable specification of π-calculus in terms of rewrite rules and prove its correctness. In a second part, we show how this specification can be used as well for real I/O, by considering interactions between a system and its environment as internal transitions of a bigger system including this environment, in our case the Unix system. A prototype written in ELAN, a rewriting interpreter [Vit94, KKM95], is working and is planned to be part of the future ELAN distributions.

This paper can be seen from two points of view. On a theoretical aspect, it is the first time up to our knowledge that an effective definition of π-calculus with explicit substitutions is given (the approach of [FMQ95], using environments, is used to study process equivalence and does not really provide an operational model). On a more practical point of view, this paper aims at showing the feasability of I/O in a rewriting context. We put the emphasis more on this second approach, and although we take great care to keep a formal link between π-calculus and the final implementation, we will not be too fussy about proofs.

2 π-calculus and its reduction relation

2.1 Why π-calculus

When choosing a calculus in which specifying explicitely input-output, we first have to decide on a conceptual model. The typical choice is to consider *processes*, able to evolve independently and to synchronize or exchange data by message passing through designated *channels*. This models fits quite well the intuition of sets of boxes connected between themselves and to the outside by some wires.

But then one may argue that π-calculus is *conceptually* too complex for such a "simple" task, and that either an ad-hoc calculus or a *conceptually* simpler calculus such as CCS with value passing or LOTOS [BB89] may do the job.

The first idea is that a special-purpose calculus expressive enough will anyway exhibit all the complexity of π-calculus, hence it is better to rely on a known calculus whose semantical foundation has been well studied. Now why π-calculus ? Is mobility of processes (the ability to dynamically change a configuration) really needed ?

The second idea is that, although introducing mobility makes a calculus more complex on a semantical point of view, *mobility comes for free* on an operational point of view. In order to *explicitely* model exchange of data, a notion of *name* (corresponding e.g. to the bound variables of λ-calculus) together with a notion of explicit substitution are needed. But then there is no difference between, say, substituting an integer value for a name, and substituting a channel name for another name. Or the other way round, we may say that value-passing comes for free when having mobility.

By acknowledging the fundamental role of names and explicit substitutions, we end up with a simpler calculus, in which mobility and exchange of data are modeled by the *same* set of rules.

2.2 The original π-calculus

In π-calculus, processes can contain bound *names* and are always considered up to α-conversion. This definition does not lead to an executable specification, because of the complexity of the axiom of α-conversion, but let us recall it anyway in order to keep a formal link between π-calculus and our implementation. Processes over a set of names $x, y, z, \ldots$ are built as follows:

$$
\begin{array}{ll}
P := \mathsf{nil} & \text{inactive process} \\
\quad \mid (\nu x)\, P & \text{restriction} \\
\quad \mid g.P & \text{guard (see below)} \\
\quad \mid P_1 + P_2 & \text{choice} \\
\quad \mid P_1 \mid P_2 & \text{parallel composition} \\
\quad \mid\, !P & \text{replication}
\end{array}
$$

There is also an *implicit* notion of substitution : $P\{y/x\}$ is the process P in which the name x has been substituted by the name y. We will introduce an explicit substitution operator later.

The g in $g.P$ stands for a *guard*. Guards are built as follows:

$$\begin{array}{ll} g := x(y) & \text{input } y \text{ on the channel } y \\ \quad \mid \overline{x}y & \text{output } y \text{ on the channel } x \\ \quad \mid \overline{x}(y) & \text{bound output} \\ \quad \mid \tau & \text{internal choice} \end{array}$$

For technical reasons, we choose to integrate bound output to the syntax. This does not however extend the calculus, since bound output is defined by the equation $\overline{x}(y) = (\nu y)\,\overline{x}y$. Note that names can be bound in three different ways: the name z is bound in $(\nu z)\,P$, $x(z).P$ and $\overline{x}(z).P$.

Processes are considered modulo the following set of structural equations: α-conversion, $\mathsf{AC}(+)$ and $\mathsf{AC}(|)$ (associativity and commutativity of the $+$ and $|$ operators), together with:

$$\begin{array}{rl} P + \mathsf{nil} \xrightarrow{=} & P \\ P \mid \mathsf{nil} \xrightarrow{=} & P \\ !P \xrightarrow{=} & P \mid !P \\ P \mid (\nu x)\,Q \xrightarrow{=} & (\nu x)\,(P \mid Q) \quad \text{if } x \text{ not free in } P \end{array}$$

The notation $\xrightarrow{=}$ will be introduced later; for the moment let us consider a rule $a \xrightarrow{=} b$ simply as an equation $a = b$. Note that we did not introduce the axiom $(\nu x)(\nu y)\,P = (\nu y)\,(\nu x)\,P$, because two processes identified by this equation have the same reduction derivations (see the definition of the Comm rule below). The equational theory entailed by these axioms is called SN, for **S**tructural equivalence of processes with **N**ames. Equality modulo SN is denoted $=_{\mathsf{SN}}$. $P/_{\mathsf{SN}}$ denotes the equivalence class of a process P modulo SN. Rewriting of classes is defined as $P/_{\mathsf{SN}} \longrightarrow Q/_{\mathsf{SN}}$ if there exist P' and Q' such that $P =_{\mathsf{SN}} P' \longrightarrow Q' =_{\mathsf{SN}} Q$.

Definition 1. A process is in weak conjunctive (resp. disjunctive) normal form if is of the form $(\nu x_1)\ldots(\nu x_k)(g_1.P_1 \mid \cdots \mid g_n.P_n)$ (resp. $(\nu x_1)\ldots(\nu x_k)(g_1.P_1 + \cdots + g_n.P_n)$), with $n \geq 0$.

2.3 Reduction relation

The reduction relation of π-calculus is given by the Comm rule (actually two rules because we have added bound output to the syntax). Note that Comm is *not* a rewrite rule, since it cannot by applied under any context:

$$\begin{array}{lll} (\cdots + x(y).P) \mid (\cdots + \overline{x}z.Q) & \xrightarrow{\mathsf{Comm}} & P\{z/y\} \mid Q \\ (\cdots + x(y).P) \mid (\cdots + \overline{x}(z).Q) & \xrightarrow{\mathsf{Comm}} & (\nu z)\,(P\{z/y\} \mid Q) \\ & & \quad \text{if } z \text{ not free in } x(y).P \end{array}$$

Reductions are allowed under parallel composition and restriction, as stated by:

$$\frac{P \xrightarrow{\mathsf{Comm}} P'}{P|Q \xrightarrow{\mathsf{Comm}} P'|Q} \qquad\qquad \frac{P \xrightarrow{\mathsf{Comm}} P'}{(\nu x)\,P \xrightarrow{\mathsf{Comm}} (\nu x)\,P'}$$

The Comm rule can however be considered as a rewrite rule if used with an appropriate strategy applying it only under the allowed contexts.

But the Comm rule is not very well suited for our purpose: it is based on the weak conjunctive normal form of processes, which does not exhibit clearly which communications are possible with an external environment.

Actually, it is possible to design a different rule based on the weak disjunctive normal form, defining the same reduction relation. This relies on the notion of reduction bisimulation:

Definition 2. A relation $\mathcal{R}$ over processes is a reduction-based bisimulation if it is reflexive and whenever $P \mathcal{R} P'$ and $P \longrightarrow Q$, then there exists Q' such that $P' \longrightarrow Q'$ and $P' \mathcal{R} Q'$. The largest reduction-based bisimulation is denoted $\sim$.

This definition states that bisimilar processes have the same derivations. Consider any equation $P = Q$ valid for reduction bisimulation. P and Q have exactly the same derivations modulo bisimilarity of processes, hence we can safely add this equation to the equational theory modulo which we consider processes. By remarking that equations valid for all common notions of bisimulation in π-calculus are also valid for reduction bisimulation [MS92, HY93], we can reuse many known results. We will consider the following two equations:

The expansion theorem [Mil91]. Let $P = \alpha_1.P_1 + ... + \alpha_n.P_n$ and $Q = \beta_1.Q_1 + ... + \beta_m.Q_m$, then

$$P \mid Q \;\xrightarrow{\sim}\; \sum_{i=1\ldots n} \alpha_i.(P_i|Q) + \sum_{i=1\ldots m} \beta_i.(P|Q_i) + \sum_{\substack{i=1\ldots n\\ j=1\ldots m}} Sync(\alpha_i.P_i, \beta_j.Q_j)$$

where

$$Sync(\alpha_i.P_i, \beta_j.Q_j) = \begin{cases} \tau.(P_i\{z/y\} \mid Q_j) \quad \text{if } \alpha_i = \mathsf{in}(x,y) \text{ and } \beta_j = \mathsf{out}(x,z) \\ (\nu z)\,\tau.(P_i\{z/y\} \mid Q_j) \;\; \text{if } \alpha_i = \mathsf{in}(x,y) \text{ and} \\ \qquad\qquad \beta_j = \mathsf{bout}(x,z) \text{ and } z \text{ not free in } P_i \\ \text{(and similarly by exchanging the arguments)} \\ \mathsf{nil} \;\; \text{in all other cases} \end{cases}$$

As before, we interpret for the moment the arrow $\xrightarrow{\sim}$ as $\sim$.

The replication theorem. Let $P = g_1.P_1 + \cdots + g_n.P_n$, then

$$!\,P \;\xrightarrow{\sim}\; g_1.(P_1 \,|\, !P) + \cdots + g_n.(P_n \,|\, !P)$$

The replication theorem is a trivial consequence of the lemma 6.3 and law L9 of [San94]. It is needed in order to put in weak disjunctive normal form processes containing the replication operator.

Let us denote SNE the equational theory formed by SN, the expansion theorem and the replication theorem. Using these equations, any process can be put in weak disjunctive normal form (we will see a constructive proof later), and we can define the counterpart of the Comm rule as

$$\tau.P + Q \xrightarrow{\mathsf{Choice}} P$$

Proposition 3.

$$P/_{\mathsf{SNE}} \xrightarrow{\mathsf{Comm}} Q/_{\mathsf{SNE}} \ \textit{iff.}\ P/_{\mathsf{SNE}} \xrightarrow{\mathsf{Choice}} Q/_{\mathsf{SNE}}$$

The implementation that we design in the following is based on the Choice rule. Note that this rule would not be suited for a semantic study of process interaction, because the information about internal synchronization is lost, but this is another topic.

2.4 Getting rid of α-conversion

As for the De Bruijn notation in the case of λ-calculus, the idea is to replace a bound name by an integer index, whose intuitive meaning is to count the numbers of binders to jump over until finding the binder corresponding to that name, and to introduce an explicit substitution operator. The difference with λ-calculus is that we have here three different kinds of binders.

The processes of π-calculus with indices are defined as follows (from now on, we adopt a syntax more digestible to a rewrite interpreter) :

- indices are integers, written always underlined, like $\underline{\mathbf{3}}$ or $\underline{\mathbf{n+1}}$.
- processes

$$\begin{array}{rll} P := & \mathsf{nil} & \text{inactive process} \\ \mid & (\nu)\,P & \text{restriction} \\ \mid & g.P & \text{guard (see below)} \\ \mid & P_1 + P_2 & \text{choice} \\ \mid & P_1 \mid P_2 & \text{parallel composition} \\ \mid & !P & \text{replication} \\ \mid & P\sigma & \text{substitution (see below)} \end{array}$$

- guards

$$\begin{array}{rll} g := & \mathsf{in}(x) & \text{input on channel } y \\ \mid & \mathsf{out}(x, y) & \text{output } y \text{ on channel } x \\ \mid & \mathsf{bout}(x) & \text{bound output} \\ \mid & \tau & \text{internal choice} \end{array}$$

- substitutions

$$\begin{array}{rll} \sigma := & [a/] & \text{where } a \text{ is an indice} \\ \mid & [\Uparrow(\sigma)] & \\ \mid & [\uparrow] & \end{array}$$

A process with indices is said well-formed if all its indices refer to an existing binder. The substitution calculus we use here is inspired by that of $\lambda\upsilon$-calculus [LRD94], chosen for its simplicity. The basic operations and their intuitive meanings are

$$\begin{array}{c} [a/] \\ \hline \underline{\mathbf{0}} \mapsto a \\ \underline{\mathbf{1}} \mapsto \underline{\mathbf{0}} \\ \dots \\ \underline{\mathbf{n+1}} \mapsto \underline{\mathbf{n}} \end{array} \qquad \begin{array}{c} [\Uparrow(s)] \\ \hline \underline{\mathbf{0}} \mapsto \underline{\mathbf{0}} \\ \underline{\mathbf{1}} \mapsto s(\underline{\mathbf{1}})[\uparrow] \\ \dots \\ \underline{\mathbf{n+1}} \mapsto s(\underline{\mathbf{n}})[\uparrow] \end{array} \qquad \begin{array}{c} [\uparrow] \\ \hline \underline{\mathbf{0}} \mapsto \underline{\mathbf{1}} \\ \underline{\mathbf{1}} \mapsto \underline{\mathbf{2}} \\ \dots \\ \underline{\mathbf{n}} \mapsto \underline{\mathbf{n+1}} \end{array}$$

The correspondance between processes with names and with indices is given by the function $\Phi(P) = \Phi_{\emptyset}(P)$ where Φ_E, defined below, is a function carrying an environment (a sequence of names) recording the bindings of variables. Note that Φ is defined only for processes with no free names (add binders if necessary).

Φ_E is defined as a congruence for all operators not involving binding and

$$\begin{array}{rcl}
\Phi_E\big((\nu x)\,P\big) & = & (\nu)\Phi_{x.E}(P) \\
\Phi_E\big(\mathsf{in}(x,y).P\big) & = & \mathsf{in}(\Phi_E(x)).\Phi_{x.E}(P) \\
\Phi_E\big(\mathsf{bout}(x,y).P\big) & = & \mathsf{bout}(\Phi_E(x)).\Phi_{x.E}(P) \\
\Phi_E(x) & = & \underline{\mathbf{i}} \;\; \text{where } i \text{ is the position of } x \text{ in } E
\end{array}$$

Most of the structural axioms are carried over without change. α-conversion is not needed any more, since it is built in the representation of indexed processes. The structural equivalence of processes with indices, SI, is given by $\mathsf{AC}(+)$, $\mathsf{AC}(\mid)$ and the following equations:

$$\begin{array}{rclrcl}
P + \mathsf{nil} & \xrightarrow{=} & P & !P & \xrightarrow{=} & P \mid !P \\
P \mid \mathsf{nil} & \xrightarrow{=} & P & P \mid (\nu)\,Q & \xrightarrow{=} & (\nu)\,(P[\uparrow] \mid Q)
\end{array}$$

The substitution calculus is defined as follows:

Congruence	Variable substitution
$\mathsf{nil}[s] \xrightarrow{=} \mathsf{nil}$	$\underline{\mathbf{0}}[a/] \xrightarrow{=} a$
$(P_1 + P_2)[s] \xrightarrow{=} P_1[s] + P_2[s]$	$\underline{\mathbf{n+1}}[a/] \xrightarrow{=} \underline{\mathbf{n}}$
$(P_1 \mid P_2)[s] \xrightarrow{=} P_1[s] \mid P_2[s]$	$\underline{\mathbf{0}}[\Uparrow(s)] \xrightarrow{=} \underline{\mathbf{0}}$
$(!P)[s] \xrightarrow{=} !(P[s])$	$\underline{\mathbf{n+1}}[\Uparrow(s)] \xrightarrow{=} \underline{\mathbf{n}}[s][\uparrow]$
$((\nu)\,P)[s] \xrightarrow{=} (\nu)\,(P[\Uparrow(s)])$	$\underline{\mathbf{n}}[\uparrow] \xrightarrow{=} \underline{\mathbf{n+1}}$
$(\mathsf{in}(x).P)[s] \xrightarrow{=} \mathsf{in}(x[s]).(P[\Uparrow(s)])$	
$(\mathsf{out}(x,y).P)[s] \xrightarrow{=} \mathsf{out}(x[s], y[s]).(P[s])$	
$(\mathsf{bout}(x).P)[s] \xrightarrow{=} \mathsf{bout}(x[s]).(P[\Uparrow(s)])$	

The replication theorem is kept unchanged. The major change is in the expansion theorem, which is the only equation introducing substitution when using the Choice rule. The expansion theorem now introduces an explicit substitution operator. It is defined as before except for:

$$Sync(\alpha_i.P_i, \beta_j.Q_j) = \begin{cases} \tau.(P_i[z/] \mid Q_j) & \text{if } \alpha_i = \mathsf{in}(x) \text{ and } \beta_j = \mathsf{out}(x,z) \\ \tau.(\nu)\,(P_i \mid Q_j) & \text{if } \alpha_i = \mathsf{in}(x) \text{ and } \beta_j = \mathsf{bout}(x) \end{cases}$$

It is interesting to note that in the case of bound output, no substitution is involved anymore. The Choice rule is kept unchanged. Let SISE be the equational theory containing SI, the substitution calculus and the expansion and replication theorems. The following propositions state the correctness of reducing processes with indices:

Proposition 4. *Φ is a bijection between processes with names (not containing free names) modulo* SNE *and (well formed) processes with indices modulo* SISE.

Proposition 5. *Let P and Q be processes with names, then*

$$P/_{\mathsf{SNE}} \xrightarrow{\mathsf{Choice}} Q/_{\mathsf{SNE}} \quad \textit{iff.} \quad \Phi(P)/_{\mathsf{SISE}} \xrightarrow{\mathsf{Choice}} \Phi(Q)/_{\mathsf{SISE}}$$

2.5 Orienting rules

Up to now, we have shown how to define the reduction relation in terms of rewriting modulo an equational theory.

But this definition does not actually provide an executable specification, because rewriting modulo such a complex theory like SISE is not realistic. However, this equational theory may itself be presented by a smaller equational theory E together with a rewrite system R convergent modulo E, and rewriting by Choice modulo SISE can be replaced by rewriting by Choice and R modulo E. This transformation and sufficient conditions guaranteeing its correctness are presented in [Vir95].

This was the idea behind the fact of writing some equations with $=$ and other with $\xrightarrow{=}$. From now on, let E be the set of non-oriented equations in SISE, namely $\mathsf{AC}(+)$ and $\mathsf{AC}(\mid)$, and R be the set of equational rules in SISE written with $\xrightarrow{=}$ or $\xrightarrow{\sim}$. (including the expansion and replication theorems), and let us try to apply the techniques of [Vir95]. We have to verify that the rewrite relation defined by R is convergent modulo E (that is, confluent and terminating), and that the rewrite relation defined by Choice is *strong coherent* with R modulo E.

The substitution calculus is confluent on ground terms [LRD94], hence over processes since we did not introduce process variables (note that there also exists substitution calculi convergent even on open terms, for instance $\lambda\sigma_{\Uparrow}$-calculus [CHL92, Les94]). All other critical pairs are easily shown to be confluent except for

$$(\nu)(P[\uparrow][\Uparrow(s)] \mid Q[\Uparrow(s)]) \;\xleftarrow{*}\; (P \mid (\nu)Q)[s] \;\xrightarrow{*}\; (\nu)(P[s][\uparrow] \mid Q[\Uparrow(s)])$$

This critical pair is shown to be confluent on ground terms by the lemma 5 used in the proof of confluence of $\lambda\upsilon$-calculus in [LRD94]. Hence we have:

Proposition 6. *R is ground confluent modulo E.*

However, R is not terminating. This is due to the replication theorem that can be applied again and again into its right-hand side. The solution is to restrict its application by a strategy Σ that basically allows an application of the replication theorem only when it is "needed" for computing a weak disjunctive normal form.

Σ is described as follows, considering a strategy as a function from terms to terms:

$$\begin{array}{rll} \Sigma(P) = & \text{if } P = \mathsf{nil} & \text{then } P \\ & \text{if } P = g.Q & \text{then } P \\ & \text{if } P = (\nu)\,Q & \text{then } (\nu)\Sigma(Q) \\ & \text{if } P = P_1 + P_2 & \text{then } \Sigma(P_1) + \Sigma(P_2) \\ & \text{if } P = P_1 \mid P_2 & \text{then } \mathit{expand}(\Sigma(P_1) \mid \Sigma(P_2)) \\ & \text{if } P = !Q & \text{then } \mathit{repl}(\Sigma(P)) \end{array}$$

where $\mathit{expand}(P)$ applies the expansion theorem at the top of P and $\mathit{repl}(P)$ the replication theorem. The remaining rules from SISE (the structural rules and the substitution rules) are applied freely. A simple induction shows that *expand* and *repl* are always applied to arguments where the corresponding rule can be applied, and the maximal number of applications of Σ is bounded by the size of the input term.

Proposition 7. *R with the strategy Σ is terminating modulo E and computes a weak disjunctive normal form.*

Finally, using the techniques of [Vir95], we establish the *strong coherence* of Choice with R by remarking that Choice is a linear rule and verifying that all critical pairs of coherence are confluent.

Proposition 8. Choice *is strong coherent with R modulo E.*

Then, using the main theorem of [Vir95], we have the following equivalence :

Proposition 9. *Let P and Q be processes with indices, then*

$$P/_{\mathsf{SISE}} \xrightarrow{\mathsf{Choice}} Q/_{\mathsf{SISE}} \quad \textit{iff.} \quad P/_{\mathsf{AC}(+,|)} \xrightarrow{\mathrm{R}}{}^{*} \xrightarrow{\mathsf{Choice}} Q'/_{\mathsf{AC}(+,|)}$$

with $Q' =_{\mathsf{SISE}} Q$

This result states that the reduction relation of π-calculus can be implemented by rewriting modulo AC, for which there are efficient interpreters or compilers available. An adequate strategy must be used for guaranteeing the termination of R and the application of Choice only under the appropriate contexts.

2.6 Value-passing and typing

So far, we have considered the plain monadic π-calculus. As described in [Mil91], this calculus is powerful enough to encode any kind of structured data, so the game may end here. However, implementing a value passing calculus based on this encoding loses one of the main advantages of rewriting logic, namely the fact of being able to combine various calculi, using for each domain the more adequate calculus without having to do unnatural encodings.

In fact, the solution is immediate. Extend the calculus by letting input and output values be not only π-calculus channels, but also any arbitrary data type (indices now range over arbitrary values, including channels). One may for instance write a process $\mathsf{in}(\underline{\mathbf{0}}).\mathsf{out}(\underline{\mathbf{1}}, 3 + \underline{\mathbf{0}}).\mathsf{nil}$, whose behaviour is intuitively "receive an integer value on a channel x ($\underline{\mathbf{0}}$), add it 3 and send the result back on the same channel (now $\underline{\mathbf{1}}$ because of the binder in)". Namely, the definition of guards is changed to

$$\begin{array}{rll} g := & \mathsf{in}(x) & \text{input } \textit{any value or indice} \text{ on the channel } x \\ | & \mathsf{out}(x, y) & \text{output } \textit{any value or indice} \text{ on the channel } x \end{array}$$

The only other change appears in the definition of the explicit substitution operator [_/], where

$\sigma := \; [a/]$ where a is an indice

is replaced by

$\sigma := \; [a/]$ where a is *any value or indice*

In order to have a conceptual difference between channels and values, some notion of *typing* is called for. This notion of typing should also be able to take into account a possible typing of values. Here we stick to the many-sorted algebra approach used in ELAN.

In the simplest option, we would have the sorts `process`, `channel` and `value`. The `channel` sort has only indices as constructors, while `values` may have also any arbitrary symbols. But indices do not fit directly in a many-sorted framework because the sort of $\underline{\mathbf{i}}$ should depend on the value i. As a workaround, we introduce a sort `indice` and a constructor C_s : `indice` $\rightarrow s$ for each sort s. All rewrite rules containing indices are duplicated for all possible C_s.

In particular, the first variable substitution rule becomes

$$C_s(\underline{\mathbf{0}})[a/] \xrightarrow{=} a \qquad \text{where } a : s$$

that is to say, it only substitutes an indice by a value of the same sort. In order to preserve the same set of normal forms, we have to ensure that this condition be always verified. The sorting discipline is able to prevent ill-sorted terms such as $C_{\texttt{channel}}(\underline{\mathbf{0}}) + 3$, but does not ensure preservation of normal forms. A sufficient condition is that all indices refering to the same binder are coerced to the same sort, what we may call well-typedness :

Proposition 10. *If P is well-typed, than the normal forms of P do not contain any substitution operator.*

Well-typedness cannot be expressed in the many-sorted framework, and in our prototype we rely on the user to provide well-typed terms. However, checking well-typedness and automatically inserting sort coercion may be performed by classical techniques borrowed from functional programming. We will not develop more on this topic here.

3 Towards a real I/O system

In the previous section, we have designed a rewrite system R (computing weak disjunctive normal forms) and a rule `Choice` modelling the reduction reduction relation of π-calculus. So far, we were still playing with a nice theoretical toy, as stressed in the introduction. Devising a real I/O system implies taking into account the physical environment, in our case the Unix system on which the rewrite interpreter is running.

This physical environment can be thought of as an unknown process E, for which we can observe the potential communications it offers to its environment. A process P able to communicate with this physical environment is thus modelled by $P \mid E$, and a physical communication by an application of the Comm rule at the top of $P \mid E$.

But it is not possible to implement this application of Comm, because we have no access to E. We can only observe the possible communications offered by E and "perform" communications using Unix system calls. "Performing" a communication will be interpreted as `read` and `write` transitions as follows:

Definition 11. If $P = \mathsf{out}(x, y).P' + U$ and $E = \mathsf{in}(x, z).E' + V$, then

$$P \xrightarrow{\mathsf{write}(x,y)} P' \quad \Longleftrightarrow \quad P \mid E \xrightarrow{\mathsf{Comm}} P' \mid E'\{y/z\}$$

If $P = \mathsf{in}(x, y).P' + U$ and $E = \mathsf{out}(x, z).E' + V$, then

$$P \xrightarrow{\mathsf{read}(x,y)} P' \quad \Longleftrightarrow \quad P \mid E \xrightarrow{\mathsf{Comm}} P'\{z/y\} \mid E'$$

Obviously, we expect the Unix `read` and `write` system calls to implement read and write transitions.

The main picture is to reduce a process P to normal form with respect to the reduction relation, then perform a read or write transition if possible, and start again. By applying the Choice rule as much as possible before attempting an external communication, it may be the case that some potentially possible communications of P will not be offered, but nevertheless the implementation is still correct with respect to the semantics of π-calculus. The only problem is that of divergent processes, for which the Choice rule may be applied infinitely often ; in this case obviously external communications should be attempted even if P is not in normal form. We leave aside this problem for future work, and suppose here that we only reduce non divergent processes.

3.1 The ELAN system

ELAN [Vit94, KKM95] is a rewrite interpreter and compiler based on many-sorted logic, capable of rewriting modulo AC and some closely related theories, with some particularly interesting features :

1. A powerful mechanism for describing strategies. A basic strategy specifies which rules may be applied at the *top* of a term and recursively the strategies to use for reducing the subterms. Strategies can be composed by choice and iteration operators. This mechanism turns out to be very expressive ; in particular, it seems that it would have been difficult to implement the strategies defined in the previous sections in a "classical" system like e.g. OBJ [GWM+92, JKKM92].
2. A powerful preprocessor for defining maps of logics. If the programmer defines a mapping from a particular calculus to rewriting logic, the user can code a program directly in this calculus, it will be transformed automatically and executed by the basic rewriting mechanism.
3. An efficiency comparable to classical programming languages for the compiled version.

The ELAN system and its documentation can be downloaded from

`ftp://ftp.loria.fr/pub/loria/protheo/softwares/Elan/`

General information can be found at

`http://www.loria.fr/equipe/protheo.html/PROJECTS/ELAN/elan.html`

3.2 Coding in ELAN

The first part of the program corresponds to the rewrite system designed in the previous section, that computes a normal form (one of the possibly many) with respect to the reduction relation (in other words performs the internal transitions).

It can be straightforwardly programmed into ELAN, except for the expansion and replication theorems which do not fit in ELAN rule syntax, because they consider terms of the form $P_1 + \cdots + P_n$ for any $n \geq 0$. The ELAN preprocessor is not of any help either, it can be used for expressing such rule schemata, but only for a bounded number of cases. The solution is to introduce auxiliary functions `expand` and `repl` that will iterate over the input term and have the effect of applying the expansion and replication theorems. The strategies for the application of Choice and termination of the replication rule are expressed in ELAN quite straightforwardly.

The system we designed is based on processes with indices, which is a representation not very readable for humans. The logic mapping capability of ELAN can be used to perform automatically the transformation between names and indices by coding the function Φ defined in section 2.4.

3.3 Unix interface

As outlined above, external communication is modelled by read and write transitions. The program we designed computes a normal form with respect to the reduction relation, of the form $P = P_1 + \cdots + P_n$, where P_i is $\mathsf{in}(x_i).Q_i$, $\mathsf{out}(x_i, y_i).Q_i$, or $\mathsf{bout}(x_i).Q_i$.

If x_i does not correspond to an external channel then P_i is not able to interact anymore, hence we simply discard P_i (ending up with a reduction-bisimilar process). If $P_i = \mathsf{bout}(x_i)$, no interaction is possible neither, because Unix does not know about channels indices and will never try to send or receive one, hence we discard P_i as well.

If x_i corresponds to an external channel (a Unix file descriptor) c, then a transition $P_i \xrightarrow{\mathsf{read}(c,z)} Q_i[z/]$ or $P_i \xrightarrow{\mathsf{write}(c,y_i)} Q_i$ is possible.

The Unix interface in ELAN is implemented as a special strategy being actually a call to an external constaint solver. We use this feature as follows. After computing a normal form of P, this special strategy is called, with the effect of sending this normal form to an external process through a Unix pipe. This process chooses among the possible input-output operations (or waits until one is possible) using the Unix `select` call, then returns the corresponding Q_i or

$Q_i[z/]$, which on the ELAN side appears as the result of the strategy (in the prototype we used a round-robin strategy in order to introduce some basic fairness in the choice between possible I/O operations). Then the same cycle is restarted from this result. If the normal form of P is nil, no input-output is requested any more and the program terminates.
Some adaptations of the program have been necessary :

(1) Since external channels correspond to Unix file descriptors, rather than maintaining a table of associations between π-calculus channels and descriptors, we choose to introduce a special constructor for external channels, $\mathsf{extchan}(i)$, where i is the file descriptor.

(2) A Unix file descriptor must be opened before its use. We introduced in the calculus a new guard for this purpose, $\mathsf{open}(filename, type, P_{Succ}, P_{Fail})$, with the intuitive meaning of opening the file whose name is given, then behave as $P_{Succ}[\mathsf{extchan}(\underline{\mathbf{i}})/]$ if the opening succeeded, binding $\underline{\mathbf{0}}$ with the corresponding channel, or behaving as $P_{Fail}[i/]$ in case of failure, binding $\underline{\mathbf{0}}$ with a system error number. The *type* argument is used to convert between ELAN and Unix data representation.

The rules for the expansion and replication theorem have been extended in order to take this new guard into account, with for instance the equation :

$$\begin{aligned} &\mathsf{open}(f, t, P_{Succ}, P_{Fail}) \mid g.Q \\ \sim\ &\mathsf{open}(f, t,\ (P_{Succ}|g.Q),\ (P_{Fail}|g.Q)) + g.(\mathsf{open}(f, t, P_{Succ}, P_{Fail}) \mid Q) \end{aligned}$$

We did not address in this stage of prototype the possibility of closing files, but this would be needed as well for practical applications.

3.4 Discussion

Programming style. A program in this framework would be based on two layers, as for instance in LOTOS [BB89], the upper one being the I/O calculus, the lower one any user rules, usually interpreted equationally. A typical program would be

$$!\,\mathsf{in}(\underline{\mathbf{0}}).\mathsf{out}(\underline{\mathbf{1}}, f(\underline{\mathbf{0}})).\mathsf{nil}$$

repeatedly reading a value x on a channel, computing $f(x)$ using the user rules, and sending the result on a channel.

It is not possible to put a process in an arbitrary context, because of the definition of the Comm rule. It may however be possible to extend this definition in some cases, namely when no non-linear rule can ever be applied in the context [Vir95]. We keep this possibility for future work.

Unix I/O semantics. The semantics of communication in π-calculus and of input-output in Unix are quite different. The former is synchronous and atomic, the latter is asynchrounous and may fail in the middle of a transfer.

Asynchronicity is actually not a problem, because it is internal to Unix : the semantic of a communication between a process P and the Unix environment is preserved. Atomicity is guaranteed when reading or writing one byte at a time.

It is possible to simulate atomicity for bigger transfers as well, but at the expense of efficiency. A real I/O system should certainly provide a choice between these both options.

Efficiency. Rewriting can be compiled into efficient programs by the ELAN compiler. An apparent source of inefficiency is the substitution calculus : applying a substitution requires a number of reductions proportional to the size of the term. However, there is no need to apply eagerly the substitution rules : particular strategies correspond to different types of known abstract machines [HMP95] and can achieve the same efficiency once compiled.

The efficiency of our prototype is currently limited by two factors : (1) We use the ELAN interpreter because AC matching is not yet implemented in the compiler. (2) Calling an external strategy from the interpreter is very expensive, as it involves a Unix process creation for each call. The ELAN compiler allows to call external code much more efficiently.

4 Conclusion

Using the notion of explicit substitutions, we designed a rewrite system whose rewriting relation modulo AC corresponds exactly to the reduction relation of π-calculus. It can be fed directly to a rewriting interpreter or compiler in order to provide an efficient implementation of this reduction relation. A few minor extensions of this calculus allow to formalize real input-output of various data types.

We hope that this input-output system will be a first step towards a wider use of rewriting-based programming frameworks, and plan to test it in real size by integrating it into the next release of the ELAN system. Some particular points still have to be developed more carefully :

(1) How to deal with non convergent processes ?

(2) "well-typedness" still relies on the user : define precisely an adequate typing system and integrate it into ELAN (which leaves room for such extensions). This should be possible by borrowing classical techniques from functional programming.

(3) Extend the calculus in order to allow to put processes in a context other than π-calculus operators. This makes little difference on an operational point of view, but is a non trivial extension to the π-calculus.

(4) Allow for recursive definitions more general than the recursion operator.

(5) Practical use of this system may suggest other useful basic constructors to integrate in the calculus.

References

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers B. V. (North-Holland), 1989.

[CHL92] P.-L. Curien, Th. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. RR 1617, INRIA, Rocquencourt, February 1992.

[Gor94] A. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994. ISBN 0 521 47103.

[GWM+92] J. A. Goguen, T. Winkler, J. Mesegeur, K. Futatsugi, and Jouannaud J.-P. Introducing OBJ. Technical Report CSL-92-03, SRI International, 1992.

[FMQ95] G.-L. Ferrari, U. Montanari, and P. Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 1995.

[HMP95] T. Hardin, L. Maranget, and B. Pagano. Functional back-ends within the weak lambda-sigma-calculus. In *Procs. of Workshop on the Implementation of Functional Languages*, Sept. 1995.

[HY93] K. Honda and N. Yoshida. On reduction-based process semantics. In *Proc. of 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 371–387. Springer-Verlag, 1993.

[JKKM92] J.-P. Jouannaud, Claude Kirchner, Hélène Kirchner, and A. Mégrelis. Programming with equalities, subsorts, overloading and parameterization in OBJ. *Journal of Logic Programming*, 12(3):257–280, February 1992.

[KKM95] C. Kirchner, H. Kirchner, and M. Vittek. Designing CLP using computational systems. In P. Van Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming.* The MIT press, 1995.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*, pages 60–69. ACM, 1994.

[LRD94] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions $\lambda\upsilon$. Technical Report RR-2222, INRIA-Lorraine, January 1994.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mil91] R. Milner. The polyadic π-calculus : a tutorial. Technical Report ECS-LFCS-91-180, LFCS, University of Edinburgh, 1991.

[MMO94] J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 906 of *LNCS*, 1994.

[MOM93] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report CSL-93-05, SRI International, 1993.

[MPW92a] Milner, Parrow, and Walker. A calculus of mobile processes pt. 1. *Information and Computation*, 100(1):1–40, Sept 1992.

[MPW92b] Milner, Parrow, and Walker. A calculus of mobile processes pt. 2. *Information and Computation*, 100(1):41–77, Sept 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Procs. of ICALP'92*, volume 623 of *LNCS*, pages 685–695. Springer-Verlag, 1992.

[San94] D. Sangiorgi. On the bisimulation proof method. Technical Report ECS-LFCS-94-299, LFCS, University of Edinburgh, 1994.

[Vir95] P. Viry. Rewriting modulo a rewrite system. Research Report RR-95-20, Dipt. di Informatica, Università di Pisa, 1995. Available from `http://www.di.unipi.it/~viry/`.

[Vit94] Marian Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, October 1994.