The rewriting calculus — Part I

HORATIU CIRSTEA, *LORIA and INRIA, Campus Scientifique,*
BP 239, 54506 Vandoeuvre-lès-Nancy, France.
E-mail: Horatiu.Cirstea@loria.fr

CLAUDE KIRCHNER, *LORIA and INRIA, Campus Scientifique,*
BP 239, 54506 Vandoeuvre-lès-Nancy, France.
E-mail: Claude.Kirchner@loria.fr

Abstract

The ρ-calculus integrates in a uniform and simple setting first-order rewriting, λ-calculus and non-deterministic computations. Its abstraction mechanism is based on the rewrite rule formation and its main evaluation rule is based on matching modulo a theory T.

In this first part, the calculus is motivated and its syntax and evaluation rules for any theory T are presented. In the syntactic case, *i.e.* when T is the empty theory, we study its basic properties for the untyped case. We first show how it uniformly encodes λ-calculus as well as first-order rewriting derivations. Then we provide sufficient conditions for ensuring confluence of the calculus.

Keywords: rewriting, strategy, non-determinism, matching, rewriting-calculus, lambda-calculus, rule based language.

1 Introduction

1.1 Rewriting, computer science and logic

It is a common claim that rewriting is ubiquitous in computer science and mathematical logic. And indeed the rewriting concept appears from the very theoretical settings to the very practical implementations. Some extreme examples are the mail system under Unix that uses rules in order to rewrite mail addresses in canonical forms (see the `/etc/sendmail.cf` file in the configuration of the mail system) and the transition rules describing the behaviors of tree automata. Rewriting is used in semantics in order to describe the meaning of programming languages [Kah87] as well as in program transformations like, for example, re-engineering of Cobol programs [vdBvDK$^+$96]. It is used in order to compute [Der85], implicitly or explicitly as in Mathematica [Wol99], MuPAD [MuP96] or OBJ [GKK$^+$87], but also to perform deduction when describing by inference rules a logic [GLT89], a theorem prover [JK86] or a constraint solver [JK91]. It is of course central in systems making the notion of rule an explicit and first class object, like expert systems, programming languages based on equational logic [O'D77], algebraic specifications (*e.g.* OBJ [GKK$^+$87]), functional programming (*e.g.* ML [Mil84]) and transition systems (*e.g.* Murphi [DDHY92]).

It is hopeless to try to be exhaustive and the cases we have just mentioned show part of the huge diversity of the rewriting concept. When one wants to focus on the underlying notions, it becomes quickly clear that several technical points should be settled. For example, what kind of objects are rewritten? Terms, graphs, strings, sets, multisets, others? Once we have established this, what is a rewrite rule? What is a left-hand side, a right-hand side, a condition, a context? And then, what is the effect of a rule application? This leads immediately to defining more technical concepts like variables in bound or free situations, substitutions and substitution application, matching, replacement; all notions being specific to the kind of objects that have to be rewritten. Once this is solved one has to understand the meaning of the application of a set of rules on (classes of) objects. And last but not least, depending on the intended use of rewriting, one would like to define an induced relation, or a logic, or a calculus.

In this very general picture, we introduce a calculus whose main design concept is to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule *application* and *result*. We concentrate on *term* rewriting, we introduce a very general notion of rewrite rule and we make the rule application and result explicit concepts. These are the basic ingredients of the *rewriting-* or *ρ-calculus* whose originality comes from the fact that terms, rules, rule application and therefore rule application strategies are all treated at the object level.

1.2 How does the rewriting calculus work?

In ρ-calculus we can explicitly represent the application of a rewrite rule, as for example $2 \to s(s(0))$, to a term, *e.g.* the constant 2, as the object $[2 \to s(s(0))](2)$ which evaluates to the singleton $\{s(s(0))\}$. This means that the rule application binary symbol "$[\_](\_)$" is part of the calculus syntax.

As we have seen a rule application can be reduced to a singleton, but it may also fail as in $[2 \to s(s(0))](3)$ that evaluates to the empty set $\emptyset$, or it can be reduced to a set with more than one element as exemplified later in this section and explained in Section 2.4. Of course, variables may be used in rewrite rules as in $[x+0 \to x](4+0)$. In this last case the evaluation mechanism of the calculus will reduce the application to $\{4\}$. In fact, when evaluating this expression, the variable x is bound to 4 via a mechanism classically called matching, and the result of the evaluation is obtained by instantiating accordingly the variable x from the right hand side of the rewrite rule. We recover, thus, the classical way term rewriting is acting.

Where this game becomes even more interesting is that "$\_ \to \_$", the rewrite binary operator, is integrally part of the calculus syntax. This is a powerful abstraction operator whose relationship with λ-abstraction [Chu40] could provide a useful intuition: A λ-expression $\lambda x.t$ can be represented in the ρ-calculus as the rewrite rule $x \to t$. Indeed, the β-redex $(\lambda x.t\ u)$ is nothing else than $[x \to t](u)$ (*i.e.* the application of the rewrite rule $x \to t$ to the term u) which reduces to $\{\{x/u\}t\}$ (*i.e.* the application of the substitution $\{x/u\}$ to the term t).

We are aware of other ways to abstract on terms or patterns in lambda-calculus e.g. the works of Colson, Kesner, van Oostrom [Col88, vO90, Kes93] or Peyton-Jones [PJ87]. For example, the λ-calculus with patterns presented in [PJ87] can be given a direct representation in the ρ-calculus. Let us consider, for example, the λ-term $\lambda(\mathit{PAIR}\ x\ y).x$ that selects the first element of a pair and the application $\lambda(\mathit{PAIR}\ x\ y).x\ (\mathit{PAIR}\ a\ b)$ that evaluates to a. The representation in the ρ-calculus of the first λ-term is $\mathit{PAIR}(x,y) \to x$ and the corresponding application $[\mathit{PAIR}(x,y) \to x](\mathit{PAIR}(a,b))$ ρ-evaluates to $\{\{x/a, y/b\}x\}$, that is to $\{a\}$.

Of course we have to make clear what a substitution $\{x/u\}$ is and how it applies to a term. But there is no surprise here and we consider a substitution mechanism that preserves the correct variable bindings via the appropriate α-conversion. In order to make this point clear in the paper, as in [DHK95], we will make a strong distinction between *substitution* (which takes care of variable binding) and *grafting* (that performs replacement directly).

When building abstractions, *i.e.* rewrite rules, there is a priori no restriction. A rewrite rule may introduce new variables as in the rule $f(x) \to g(x,y)$ that when applied to the term $f(a)$ evaluates to $\{g(a,y)\}$, leaving the variable y free. It may also rewrite an object into a rewrite rule as in the application $[x \to (f(y) \to g(x,y))](a)$ that evaluates to the singleton $\{f(y) \to g(a,y)\}$. In this case the variable x is free in the rewrite rule $f(y) \to g(x,y)$ but is bound in the rule $x \to (f(y) \to g(x,y))$. More generally, the object formation in ρ-calculus is unconstrained. Thus, the application of the rule $b \to c$ after the rule $a \to b$ to the term a is written $[b \to c]([a \to b](a))$ and as expected the evaluation mechanism will produce first $[b \to c](\{b\})$ and then $\{c\}$. It also allows us to make use in an explicit and direct way of non-terminating or non-confluent (equational) rewrite systems. For example the application of the rule $a \to a$ to the term a ($[a \to a](a)$) terminates, since it is applied only once and does not represent the *repeated* application of the rewrite rule $a \to a$.

So, basic ρ-calculus objects are built from a signature, a set of variables, the abstraction operator "$\to$", the application operator "[ ]( )", and we consider sets of such objects. This gives to the ρ-calculus the ability to handle non-determinism in the sense of sets of results. This is achieved via the explicit handling of reduction result sets, including the empty set that records the fundamental information of rule application failure. For example, if the symbol $+$ is assumed to be commutative then $x + y$ is equivalent modulo commutativity to $y + x$ and thus applying the rule $x+y \to x$ to the term $a+b$ results in $\{a, b\}$. Since there are two different ways to apply (match) this rewrite rule modulo commutativity the result is a set that contains two different elements corresponding to two possibilities. This ability to integrate specific computations in the matching process allows us for example to use the ρ-calculus for deduction modulo purposes as proposed in [DHK98].

To summarize, in ρ-calculus abstraction is handled via the arrow binary operator, matching is used as the parameter passing mechanism, substitution takes care of variable bindings and results sets are handled explicitly.

1.3 Rewriting relation versus rewriting calculus

A ρ-calculus term contains all the (rewrite rule) information needed for its evaluation. This is also the case for λ-calculus but it is quite different from the usual way term rewrite *relations* are defined.

The rewrite relation generated by a rewrite system $\mathcal{R} = \{l_1 \to r_1, \ldots, l_n \to r_n\}$ is defined as the smallest transitive relation stable by context and substitution and containing $(l_1, r_1), \ldots, (l_n, r_n)$. For example if $\mathcal{R} = \{a \to f(a)\}$, then the rewrite relation contains $(a, f(a))$, $(a, f(f(a)))$, $(f(a), f(f(a))), \ldots$ and one says that the derivation $a \to f(a) \to f(f(a)) \to \ldots$ is generated by $\mathcal{R}$.

In ρ-calculus the situation is different since ρ-evaluation will reduce a given ρ-term in which all the rewriting information is explicit. It is customary to say that the rewrite system $a \to a$ is not terminating because it generates the derivation $a \to a \to a \to \ldots$. In ρ-calculus the same infinite derivation should be explicitly built (for example using an iterator) and all the evaluation information should be present in the starting term as in $[a \to a]([a \to a]([a \to a](a)))$ whose evaluation corresponds to the three steps derivation $a \to a \to a \to a$.

There is thus a big difference between the way one can define rewrite derivations generated by a rewrite system and their representation in ρ-calculus: in the first case the derivation construction is implicit and left at the meta-level, in the later case, all rewrite steps should be explicitly built.

1.4 Integration of first-order rewriting and higher-order logic

We are introducing a new calculus in a heavily-charged landscape. Why one more? There are several complementary answers that we will make explicit in this work. One of them is the unifying principle of the calculus with respect to algebraic and higher-order theories.

The integration of first-order and higher-order paradigms has been one of the main problems raised since the beginning of the study of programming language semantics and of proof environments. The λ-calculus emerged in the thirties and had a deep influence on the development of theoretical computer-science as a simple but powerful tool for describing programming language semantics as well as proof development systems. Term rewriting for its part emerged as an identified concept in the late sixties and it had a deep influence in the development of algebraic specifications as well as in theorem proving.

Because the two paradigms have a lot in common but have extremely useful complementary properties, many works address the integration of term rewriting with λ-calculus. This has been handled either by enriching first-order rewriting with higher-order capabilities or by adding to λ-calculus algebraic features allowing one, in particular, to deal with equality in an efficient way. In the first case, we find the works on CRS [KvOvR93], XRS [Pag98] and other higher-order rewriting systems [Wol93, NP98], in the second case the works on combination of λ-calculus with term rewriting [Oka89, BT88, GBT89, JO97] to mention only a few.

Our previous works on the control of term rewriting [KKV95, Vit94, BKKR01] led us to introduce the ρ-calculus. Indeed we realized that the tool that is needed in order to control rewriting should be made explicit and could be itself naturally described using rewriting. By viewing the arrow rewrite symbol as an abstraction operator, we strictly generalize the abstraction mechanism of λ-calculus, by making the rule application explicit, we get full control of the rewrite mechanism and as a consequence we obtain with the ρ-calculus a uniform integration of algebraic computation and λ-calculus.

1.5 Basic properties and uses of the ρ-calculus

One of the main properties of the calculus we are concentrating on is confluence. We will see that the ρ-calculus is not confluent in the general case. The use of sets for representing the reduction results is the main cause of non-confluence. This comes from the fact that in the definition of a standard rewrite step, a rule is applied only when a successful match is found and in this case the reduced term exists and is unique (even if several matches exist). In ρ-calculus we are in a very different situation since a rule application always yields a unique result consisting either of a non-empty set representing all the possible reduced terms (one per different match) or of an empty set representing the impossibility to apply a standard rewrite step.

The confluence can be recovered if the evaluation rules of ρ-calculus are guided by an appropriate strategy. This strategy should first handle properly the problems related to the propagation of failure over the operators of the calculus. It should also take care of the correct handling of sets with more than one element in non-linear contexts. We are presenting this strategy whose full details are given in [Cir00].

We will see that the ρ-calculus can be used for representing some simpler calculi as λ-calculus and rewriting even in the conditional case. This is achieved by restricting the syntax and the evaluation rules of the ρ-calculus in order to represent the terms of the two calculi. We then show that for any reduction in the λ-calculus or term rewriting, a corresponding natural reduction in the ρ-calculus can be found.

1.6 Structure of this work and paper

The presentation of this work is divided in two parts, the second one being called hereafter *Part II.*

The purpose of this first part is to introduce the ρ-calculus, its syntax and evaluation rules and to show how it can be used in order to naturally encode λ-calculus and standard term rewriting. We also show in *Part II*, and indeed this was our first motivation, that it can be used to encode conditional rewriting and that it provides a semantics for the rewrite based language ELAN.

In the next section, we introduce the general ρ_T-calculus, where T is a theory used to internalize specific knowledge like associativity and commutativity of certain operators. We present the syntax of the calculus, its evaluation rules together with examples. We emphasize in particular the important role of the matching theory T. We show in Section 3 how ρ-calculus can be used to encode in a uniform way term rewriting and λ-calculus. Then, in Section 4, we restrict to the $\rho_\emptyset$-calculus (also shortly denoted ρ-calculus), the calculus where only syntactic matching is allowed (*i.e.* the theory T is assumed to be the trivial one), and we present the confluence properties of this calculus. We assume the reader familiar with the standard notions of term rewriting [DJ90, Klo90, BN98, KK99] and with the basic notions of λ-calculus [Bar84]. For the basic concepts about rule based constraint solving and *deduction modulo*, we refer respectively to [JK91, KR98] and [DHK98].

2 Definition of the ρ_T-calculus

We assume given in this section a theory T defined equationally or by any other means.

A calculus is defined by the following five components:

1. First its *syntax* that makes precise the formation of the objects manipulated by the calculus as well as the formation of substitutions that are used by the evaluation mechanism. In the case of ρ_T-calculus, the core of the object formation relies on

a first-order signature together with rewrite rules formation, rule application and sets of results.

2. The description of the *substitution application* to terms. This description is often given at the meta-level, except for explicit substitution frameworks. For the description of the ρ_T-calculus that we give here, we use (higher-order) substitutions and not grafting, *i.e.* the application takes care of variable bindings and therefore uses α-conversion.
3. The *matching algorithm* used to bind variables to their actual values. In the case of ρ_T-calculus, this is matching modulo the theory T. In practical cases it will be higher-order-pattern matching, or equational matching, or simply syntactic matching or combination of any of these. The matching theory is specified as a parameter (the theory T) of the calculus and when it is clear from the context this parameter is omitted.
4. The *evaluation rules* describing the way the calculus operates. It is the glue between the previous components. The simplicity and clarity of these rules are fundamental for its usability.
5. The *strategy* guiding the application of the evaluation rules. Depending on the strategy employed we obtain different versions and therefore different properties for the calculus.

This section makes explicit all these components for the ρ_T-calculus and comments our main choices.

2.1 Syntax of the ρ_T-calculus

Definition 2.1
We consider $\mathcal{X}$ a set of variables and $\mathcal{F} = \bigcup_m \mathcal{F}_m$ a set of ranked function symbols, where for all m, $\mathcal{F}_m$ is the subset of function symbols of arity m. We assume that each symbol has a unique arity *i.e.* that the $\mathcal{F}_m$ are disjoint. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the set of first-order terms built on $\mathcal{F}$ using the variables in $\mathcal{X}$. The set of basic ρ-terms, denoted $\varrho(\mathcal{F}, \mathcal{X})$, is the smallest set of objects formed according to the following rules:

- the variables in $\mathcal{X}$ are ρ-terms,
- if $t_1, \ldots, t_n$ are ρ-terms and $f \in \mathcal{F}_n$ then $f(t_1, \ldots, t_n)$ is a ρ-term,
- if $t_1, \ldots, t_n$ are ρ-terms then $\{t_1, \ldots, t_n\}$ is a ρ-term (the empty set is denoted $\emptyset$),
- if t and u are ρ-terms then $[t](u)$ is a ρ-term (application),
- if t and u are ρ-terms then $t \rightarrow u$ is a ρ-term (abstraction or rewrite rule).

The set of basic ρ-terms can thus be inductively defined by the following grammar:

$$\rho\text{-terms} \quad t \quad ::= \quad x \mid f(t, \ldots, t) \mid \{t, \ldots, t\} \mid [t](t) \mid t \rightarrow t$$

where $x \in \mathcal{X}$ and $f \in \mathcal{F}$. Notice that this syntax does not make use of the theory T.

A term may be viewed as a *finite labeled ordered tree*, the leaves of which are labeled with variables or constants and the internal nodes of which are labeled with symbols of positive arity.

Definition 2.2

A *position* (also called *occurrence*) of a term (seen as a tree) is represented as a sequence ω of positive integers describing the path from the root of t to the root of the sub-term at that position. We denote by $t_{\lceil s \rceil_p}$ the term t containing the sub-term s at the position p. The symbol at the position p of a term t is denoted by $t(p)$.

We call *functional position* of a ρ-term t, any occurrence p of the term whose symbol belongs to $\mathcal{F}$, *i.e.* $t(p) \in \mathcal{F}$. The set of all positions of a term t is denoted by $\mathcal{P}os(t)$. The set of all functional positions of a term t is denoted by $\mathcal{FP}os(t)$.

The position of a sub-term in a set ρ-term is obtained by considering one of the possible tree representations of the respective ρ-term.

We adopt a very general discipline for the rewrite rule formation, and we do not enforce any of the standard restrictions often used in the term rewriting community like non-variable left-hand sides or occurrence of the right-hand side variables in the left-hand side. We also consider rewrite rules containing rewrite rules as well as rewrite rule application. For convenience, we consider that the symbols $\{\}$ and $\emptyset$ both represent the empty set. We usually use the notation f instead of $f()$ for a function symbol of arity 0 (*i.e.* a constant). For the terms of the form $\{t_1, \ldots, t_n\}$ we assume, as usually, that the comma is an associative, commutative and idempotent function symbol.

The main intuition behind this syntax is that a rewrite rule is an abstraction, the left-hand side of which determines the bound variables and some contextual information. Having new variables in the right-hand side is just the ability to have free variables in the calculus. We will come back to this later but to support the intuition let us mention that the λ-terms [Bar84] and standard first-order rewrite rules [DJ90, BN98] are clearly objects of this calculus. For example, the λ-term $\lambda x.(y\ x)$ corresponds to the ρ-term $x \to [y](x)$ and a rewrite rule in first-order rewriting corresponds to the same rewrite rule in the rewriting-calculus.

We have chosen sets as the data structure for handling the potential non-determinism. A set of terms can be seen as the set of distinct results obtained by applying a rewrite rule to a term. Other choices could be made depending on the intended use of the calculus. For example, if we want to provide all the results of an application, including the identical ones, a multi-set could be used. When the order of the computation of the results is important, lists could be employed. Since in this presentation of the calculus we focus on the possible results of a computation and not on their number or order, sets are used. The confluence properties presented in Section 4 are preserved in a multi-set approach. It is clear that for the list approach only a confluence modulo permutation of lists can be obtained.

The following examples show the very expressive syntax that is allowed for ρ-terms.

Example 2.3

If we consider $\mathcal{F}_0 = \{a, b, c\}$, $\mathcal{F}_1 = \{f\}$, $\mathcal{F}_2 = \{g\}$, $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2$ and x, y variables in $\mathcal{X}$, some ρ-terms from $\varrho(\mathcal{F}, \mathcal{X})$ are:

- $[a \to b](a)$; this denotes the application of the rewrite rule $a \to b$ to the term a. We will see that evaluating this application results in $\{b\}$.
- $[g(x, y) \to f(x)](g(a, b))$; a classical rewrite rule application.
- $[x \to x + y](a)$; a rewrite rule with a free variable y. We will see later why the result of this application is $\{a + y\}$ where the variable y remains free.

- $[y \to [x \to x + y](b)]([x \to x](a))$; a ρ-term that corresponds to the λ-term $(\lambda y.((\lambda x.x + y)\ b))\ ((\lambda x.x)\ a)$. In the rewrite rule $x \to x + y$ the variable y is free but in the rewrite rule $y \to [x \to x + y](b)$ this variable is bound.
- $[x \to [x](x)](x \to [x](x))$; the well-known $(\omega\omega)$ λ-term. We will see that the evaluation of this term is not terminating.
- $[[(x \to x + 1) \to (1 \to x)](a \to a + 1)](1)$; a more complicated ρ-term without corresponding standard rewrite rule or λ-term.

2.2 Grafting versus substitution

Since we are dealing with $\to$ as a binder, like for any calculus involving binders (as the λ-calculus), α-conversion should be used to obtain a correct substitution calculus and the first-order substitution (called here grafting) is not directly suitable for the ρ-calculus. We consider the usual notions of α-conversion and higher-order substitution as defined for example in [DHK95].

This is the reason for introducing an appropriate notion of bound variables renaming in Definition 2.5. It computes a variant of a ρ-term which is equivalent modulo α-conversion to the initial term.

Definition 2.4
The set of free variables of a ρ-term t is denoted by $FV(t)$ and is defined by:

1. if $t = x$ then $FV(t) = \{x\}$,
2. if $t = f(u_1, \ldots, u_n)$ then $FV(t) = \bigcup_{i=1,\ldots,n} FV(u_i)$,
3. if $t = \{u_1, \ldots, u_n\}$ then $FV(t) = \bigcup_{i=1,\ldots,n} FV(u_i)$,
4. if $t = [u](v)$ then $FV(t) = FV(u) \cup FV(v)$,
5. if $t = u \to v$ then $FV(t) = FV(v) \setminus FV(u)$.

Definition 2.5
Given a set $\mathcal{Y}$ of variables, the application $\alpha_{\mathcal{Y}}$ (called α-conversion) is defined by:

- $\alpha_{\mathcal{Y}}(x) = x$,
- $\alpha_{\mathcal{Y}}(f(u_1, \ldots, u_n)) = f(\alpha_{\mathcal{Y}}(u_1), \ldots, \alpha_{\mathcal{Y}}(u_n))$,
- $\alpha_{\mathcal{Y}}(\{t\}) = \{\alpha_{\mathcal{Y}}(t)\}$,
- $\alpha_{\mathcal{Y}}([t](u)) = [\alpha_{\mathcal{Y}}(t)](\alpha_{\mathcal{Y}}(u))$,
- $\alpha_{\mathcal{Y}}(u \to v) = \alpha_{\mathcal{Y}}(u) \to \alpha_{\mathcal{Y}}(v)$, if $FV(u) \cap \mathcal{Y} = \emptyset$,
- $\alpha_{\mathcal{Y}}(u \to v) = (\{x_i \mapsto y_i\}_{x_i \in FV(u)}\ \alpha_{\mathcal{Y}}(u)) \to (\{x_i \mapsto y_i\}_{x_i \in FV(u)}\ \alpha_{\mathcal{Y}}(v))$, if $x_i \in FV(u) \cap \mathcal{Y}$ and y_i are "fresh" variables and where $\{x \mapsto y\}$ denotes the replacement of the variable x by the variable y in the term on which it is applied.

This allows us to define the usual substitution and grafting operations:

Definition 2.6
A *valuation* θ is a finite binding of the variables $x_1, \ldots, x_n$ to the terms $t_1, \ldots, t_n$, *i.e.* a finite set of couples $\{(x_1, t_1), \ldots, (x_n, t_n)\}$.

From a given valuation θ we can define the following two notions of substitution and grafting:

- the *substitution* extending θ is denoted $\Theta = \{x_1/t_1, \ldots, x_n/t_n\}$,

- the *grafting* extending θ is denoted $\bar{\Theta} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.

Θ and $\bar{\Theta}$ are structurally defined by:

– $\Theta(x) = u$, if $(x, u) \in \theta$
– $\Theta(f(t_1 \ldots t_n)) = f(\Theta(t_1) \ldots \Theta(t_n))$
– $\Theta(\{t_1, \ldots, t_n\}) = \{\Theta(t_1), \ldots, \Theta(t_n)\}$
– $\Theta([t](u)) = [\Theta(t)](\Theta(u))$
– $\Theta(u \to v) = \Theta(u') \to \Theta(v')$

– $\bar{\Theta}(x) = u$, if $(x, u) \in \theta$
– $\bar{\Theta}(f(t_1 \ldots t_n)) = f(\bar{\Theta}(t_1) \ldots \bar{\Theta}(t_n))$
– $\bar{\Theta}(\{t_1, \ldots, t_n\}) = \{\bar{\Theta}(t_1), \ldots, \bar{\Theta}(t_n)\}$
– $\bar{\Theta}([t](u)) = [\bar{\Theta}(t)](\bar{\Theta}(u))$
– $\bar{\Theta}(u \to v) = \bar{\Theta}(u) \to \bar{\Theta}(v)$

where we consider that z_i are fresh variables (*i.e.* $\theta z_i = z_i$), the z_i do not occur in u and v and for any $y \in FV(u)$, $z_i \notin FV(\theta y)$, and u', v' are defined by:
$u' = \{y_i \mapsto z_i\}_{y_i \in FV(u)} \; \alpha_{FV(u) \cup \mathcal{V}ar(\theta)}(u)$,
$v' = \{y_i \mapsto z_i\}_{y_i \in FV(u)} \; \alpha_{FV(u) \cup \mathcal{V}ar(\theta)}(v)$.
using the following notations: The set of variables $\{x_1, \ldots, x_n\}$ is called the domain of the substitution Θ or of the grafting $\bar{\Theta}$ and is denoted by $\mathcal{D}om(\Theta)$ or $\mathcal{D}om(\bar{\Theta})$ respectively. The set of all the variables from Θ is $\mathcal{V}ar(\Theta) = \cup_{x \in \mathcal{D}om(\Theta)} \Theta(x) \cup \mathcal{D}om(\Theta)$.

Recall that $\{x_1/t_1, \ldots, x_n/t_n\}$ is the simultaneous substitution of the variables $x_1, \ldots, x_n$ by the terms $t_1, \ldots, t_n$ and not the composition $\{x_1/t_1\} \ldots \{x_n/t_n\}$.

There is nothing new in the definition of substitution and grafting except that the abstraction works here on terms and not only on variables. The burden of variable handling could be avoided by using an explicit substitution mechanism in the spirit of [CHL96]. We sketched such an approach in [CK99] and this is detailed in [Cir00].

2.3 Matching

Computing the matching substitutions from a ρ-term t to a ρ-term t' is an important parameter of the ρ_T-calculus. We first define matching problems in a general setting:

Definition 2.7
For a given theory T over ρ-terms, a *T-match-equation* is a formula of the form $t \ll^?_T t'$, where t and t' are ρ-terms. A substitution σ is a solution of the T-match-equation $t \ll^?_T t'$ if $T \models \sigma(t) = t'$. A *T-matching system* is a conjunction of T-match-equations. A substitution is a solution of a T-matching system P if it is a solution of all the T-match-equations in P. We denote by $\mathbf{F}$ a T-matching system without solution. A T-matching system is called *trivial* when all substitutions are solution of it.
We define the function *Solution* on a T-matching system $\mathcal{S}$ as returning the set of all T-matches of $\mathcal{S}$ when $\mathcal{S}$ is not trivial and $\{\mathbb{ID}\}$, where $\mathbb{ID}$ is the identity substitution, when $\mathcal{S}$ is trivial.

Notice that when the matching system has no solution the function *Solution* returns the empty set.

Since in general we could consider arbitrary theories over ρ-terms, T-matching is in general undecidable, even when restricted to first-order equational theories [JK91]. In order to overcome this undecidability problem, one can think of using constraints as in constrained higher-order resolution [Hue73] or constrained deduction [KKR90]. But we are interested here in the decidable cases. Among them we can mention higher-order-pattern matching that is decidable and unitary as a consequence of the decidability of pattern unification [Mil91, DHKP96], higher-order matching which is known to be decidable up to the fourth order [Pad96, Pad00, Dow94, HL78] (the decidability of the general case being still open), many first-order equational theories including associativity, commutativity, distributivity and most of their combinations [Nip89, Rin96].

For example when T is empty, the syntactic matching substitution from t to t', when it exists, is unique and can be computed by a simple recursive algorithm given for example by G. Huet [Hue76]. It can also be computed by the following set of rules *SyntacticMatching* where $f, g \in \mathcal{F}$ and the symbol $\wedge$ is assumed to be associative and commutative.

Decomposition	$(f(t_1,\ldots,t_n) \ll^?_\emptyset f(t'_1,\ldots,t'_n)) \wedge P$	$\mapstochar\twoheadrightarrow$	$\bigwedge_{i=1\ldots n} t_i \ll^?_\emptyset t'_i \wedge P$
SymbolClash	$(f(t_1,\ldots,t_n) \ll^?_\emptyset g(t'_1,\ldots,t'_m)) \wedge P$	$\mapstochar\twoheadrightarrow$	$\mathbf{F}$ if $f \neq g$
MergingClash	$(x \ll^?_\emptyset t) \wedge (x \ll^?_\emptyset t') \wedge P$	$\mapstochar\twoheadrightarrow$	$\mathbf{F}$ if $t \neq t'$
VariableClash	$(f(t_1,\ldots,t_n) \ll^?_\emptyset x) \wedge P$	$\mapstochar\twoheadrightarrow$	$\mathbf{F}$ if $x \in \mathcal{X}$

FIG. 1. *SyntacticMatching* - Rules for syntactic matching

PROPOSITION 2.8
The normal form by the rules in *SyntacticMatching* of any matching problem $t \ll^?_\emptyset t'$ exists and is unique. After removing from the normal form any duplicated match-equation and the trivial match-equations of the form $x \ll^?_\emptyset x$ for any variable x, if the resulting system is:

1. $\mathbf{F}$, then there is no match from t to t' and $\mathcal{S}olution(t \ll^?_\emptyset t') = \mathcal{S}olution(\mathbf{F}) = \emptyset$,
2. of the form $\bigwedge_{i\in I} x_i \ll^?_\emptyset t_i$ with $I \neq \emptyset$, then the substitution $\sigma = \{x_i/t_i\}_{i\in I}$ is the unique match from t to t' and $\mathcal{S}olution(t \ll^?_\emptyset t') = \mathcal{S}olution(\bigwedge_{i\in I} x_i \ll^?_\emptyset t_i) = \{\sigma\}$,
3. empty, then t and t' are identical and $\mathcal{S}olution(t \ll^?_\emptyset t) = \{\mathbb{ID}\}$.

PROOF. See [KK99].

□

EXAMPLE 2.9
If we consider the matching problem $(h(x, g(x, y)) \ll^?_\emptyset h(a, g(a, b))$, first we apply the matching rule *Decomposition* and we obtain the system with the two match-equations $(x \ll^?_\emptyset a)$ and $(g(x, y) \ll^?_\emptyset g(a, b))$. When we apply the same rule once again for the second equation we obtain $(x \ll^?_\emptyset a)$ and $(y \ll^?_\emptyset b)$ and thus, the initial match-equation is reduced to the system $(x \ll^?_\emptyset a) \wedge (x \ll^?_\emptyset a) \wedge (y \ll^?_\emptyset b)$ and $\mathcal{S}olution(h(x, g(x, y)) \ll^?_\emptyset h(a, g(a, b)) = \{\{x/a, y/b\}\}$.

For the matching problem $(g(x, x) \ll^?_\emptyset g(a, b))$ we apply, as before, *Decomposition* and we obtain the system $(x \ll^?_\emptyset a) \wedge (x \ll^?_\emptyset b)$. This latter system is reduced by the matching rule *MergingClash* to $\mathbf{F}$ and thus, $\mathcal{S}olution(g(x, x) \ll^?_\emptyset g(a, b)) = \emptyset$.

This syntactic matching algorithm has an easy and natural extension when a symbol $+$ is assumed to be commutative. In this case, the previous set of rules should be completed with

$$\begin{array}{ll} \mathit{CommDec} & (t_1 + t_2) \ll^?_{C(+)} (t'_1 + t'_2) \wedge P \;\; \mapsto\!\!\!\twoheadrightarrow \\ & ((t_1 \ll^?_{C(+)} t'_1 \wedge t_2 \ll^?_{C(+)} t'_2) \vee (t_1 \ll^?_{C(+)} t'_2 \wedge t_2 \ll^?_{C(+)} t'_1)) \wedge P \end{array}$$

where disjunction should be handled in the usual way. In this case of course the number of matches could be exponential in the size of the initial left-hand sides.

Example 2.10
When matching modulo commutativity the term $x{+}y$, with $+$ defined as commutative, against the term $a + b$, the rule *CommDec* leads to

$$((x \ll^?_{C(+)} a \wedge y \ll^?_{C(+)} b) \vee (x \ll^?_{C(+)} b \wedge y \ll^?_{C(+)} a))$$

and thus, we obtain two substitutions as solution for the initial matching problem, *i.e.* $\mathit{Solution}(x + y \ll^?_{C(+)} a + b) = \{\{x/a, y/b\}, \{x/b, y/a\}\}$.

Matching modulo associativity-commutativity (AC) is often used. It could be defined either in a rule based way as in [AK92, KR98] or in a semantic way as in [Eke95]. A restricted form of associative matching called *list matching* is used in the ASF+SDF system [vD96]. In the Maude system any combination of the associative, commutative and idempotency properties is available [Eke96].

2.4 Evaluation rules of the ρ_T-calculus

Assume we are given a theory T over ρ-terms having a decidable matching problem. The use of constraints would allow us to drop this last restriction, but we have chosen here to stick to this simpler situation.

As mentioned above, in the general case, the matching is not unitary and thus we should deal with (empty, finite or infinite) sets of substitutions. We consider a substitution application at the meta-level of the calculus represented by the operator “$\_\langle\!\langle \_ \rangle\!\rangle$” whose behavior is described by the meta-rule *Propagate*:

$$\mathit{Propagate} \quad r\langle\!\langle \{\sigma_1, \ldots, \sigma_n, \ldots\} \rangle\!\rangle \;\; \rightsquigarrow \;\; \{\sigma_1 r, \ldots, \sigma_n r, \ldots\}$$

Notice that since this rule operates at the meta-level of the calculus, it is different from the evaluation rules like *Fire* and its arrow is denoted differently. A version of the calculus can also be given using explicit substitution [Cir00].

The result of the application of a set of substitutions $\{\sigma_1, \ldots, \sigma_n, \ldots\}$ to a term r is the set of terms $\sigma_i r$, where $\sigma_i r$ represents the result of the (meta-)application of the substitution σ_i to the term r as detailed in Definition 2.6. Notice that when n is 0, *i.e.* the set of substitutions is empty, the resulting set of instantiated terms is also empty.

The evaluation rules of the ρ_T-calculus describe the application of a ρ-term on another one and specify the behavior of the different operators of the calculus when some arguments are sets. Following their specifications they are described in Figure 2 to 5.

2.4.1 Applying rewrite rules

The application of a rewrite rule at the root position of a term is accomplished by matching the left-hand side of the rewrite rule on the term and returning the appropriately instantiated right-hand side. It is described by the evaluation rule *Fire* in Figure 2. The rule *Fire*, like all the evaluation rules of the calculus, can be applied at any position of a ρ-term.

$$\boxed{\quad \mathit{Fire} \quad [l \to r](t) \quad \Longrightarrow \quad r\langle\!\langle \mathcal{S}olution(l \ll^?_T t)\rangle\!\rangle \quad}$$

FIG. 2. The evaluation rule *Fire* of the ρ_T-calculus

The central idea is that applying a rewrite rule $l \to r$ at the root (also called top) occurrence of a term t, written as $[l \to r](t)$, consists in replacing the term r by $r\langle\!\langle \Sigma \rangle\!\rangle$ where Σ is the set of substitutions obtained by T-matching l on t (*i.e.* $\mathcal{S}olution(l \ll^?_T t)$). Therefore, when the matching yields a failure represented by an empty set of substitutions, the result of the application of the rule *Propagate* and thus of the rule *Fire* is the empty set.

One can notice that the rule *Fire* can be expressed without using the meta-rule *Propagate*:

$$\begin{array}{lll} \mathit{Fire} \quad [l \to r](t) & \rightsquigarrow & \{\sigma_1 r, \ldots, \sigma_n r, \ldots\} \\ & & \text{where } \{\sigma_1, \ldots, \sigma_n, \ldots\} = \mathcal{S}olution(l \ll^?_T t) \end{array}$$

but we preferred the previous version for a smoother transition to the explicit version of the calculus.

We should point out that, as in λ-calculus, an application can always be evaluated. But, unlike in λ-calculus, the set of results can be empty. More generally, when matching modulo a theory T, the set of resulting matches may be empty, a singleton (as in the empty theory), a finite set (as for associativity-commutativity) or infinite (see [FH83]). We have thus chosen to represent the result of a rewrite rule application to a term as a set. An empty set means that the rewrite rule $l \to r$ fails to apply to t in the sense of a matching failure between l and t.

We denote by $\longrightarrow_{Fire}$ the relation induced by the evaluation rule *Fire*.

EXAMPLE 2.11
Some examples of the application of the evaluation rule *Fire* are:

- $[a \to b](a) \longrightarrow_{Fire} \{b\}$
- $g(x, [x \to c](a)) \longrightarrow_{Fire} g(x, \{c\})$
- $[a \to b](c) \longrightarrow_{Fire} \emptyset$

2.4.2 Applying operators

In order to push rewrite rule application deeper into terms, we introduce the two *Congruence* evaluation rules of Figure 3. They deal with the application of a term of the form $f(u_1, \ldots, u_n)$ (where $f \in \mathcal{F}_n$) to another term of a similar form. When we have the same head symbol for the two terms of the application $[u](v)$ the arguments of the term u are applied on those of the term v argument-wise. If the head symbols are not the same, an empty set is obtained.

$$
\begin{array}{lllll}
\mathit{Cong} & [f(u_1,\ldots,u_n)](f(v_1,\ldots,v_n)) & \Longrightarrow & \{f([u_1](v_1),\ldots,[u_n](v_n))\} \\
\mathit{CongFail} & [f(u_1,\ldots,u_n)](g(v_1,\ldots,v_m)) & \Longrightarrow & \emptyset
\end{array}
$$

FIG. 3. The evaluation rules *Congruence* of the ρ_T-calculus

REMARK 2.12
The *Congruence* rules are redundant with respect to the evaluation rule *Fire* modulo an appropriate transformation of the initial term. Indeed, one could notice that the application of a term $f(u_1,\ldots,u_n)$ to another ρ-term t (*i.e.* the ρ-term $[f(u_1,\ldots,u_n)](t)$) evaluates, using the rules *Cong* and *CongFail*, to the same term as the application of the ρ-term $f(x_1,\ldots,x_n) \to f([u_1](x_1),\ldots,[u_n](x_n))$ on the same term t (*i.e.* the ρ-term $[f(x_1,\ldots,x_n) \to f([u_1](x_1),\ldots,[u_n](x_n))](t)$) using the evaluation rule *Fire*. Although we can express the same computations by using only the evaluation rule *Fire*, we prefer to keep the evaluation rules *Congruence* in the calculus for an explicit use of these rules and thus, a more concise representation of terms.

2.4.3 Handling sets in the ρ_T-calculus

The reductions describing the behavior of terms containing sets are described by the evaluation rules in Figure 4:

- The rules *Distrib* and *Batch* describe the interaction between the application and the set operators,
- The rules Switch_L and Switch_R describe the interaction between the abstraction and the set operators,
- The rule *OpOnSet* describe the interaction between the symbols of the signature and the set operators.
- The rule describing the interaction between set operators will be described in the next section.

The set representation for the results of the rewrite rule application has important consequences concerning the behavior of the calculus. We can notice, in particular, that the number of set symbols is unchanged by the evaluation rules *Distrib*, *Batch*, Switch_L, Switch_R and *OpOnSet*. This way, for a derivation involving only terms that do not contain empty sets, the number of set symbols in a term counts the number of rules *Fire* and *Congruence* that have been applied for its evaluation.

The application of the set of rewrite rules $\{a \to b, a \to c\}$ to the term a (*i.e.* the ρ-term $[\{a \to b, a \to c\}](a)$) is reduced, by using the evaluation rule *Distrib*, to the set containing the application of each rule to the term a (*i.e.* the ρ-term $\{[a \to b](a), [a \to c](a)\}$). It is in particular useful when simulating ordinary term rewriting by a *set* of rewrite rules. Moreover, we can factor a set of rewrite rules

$$
\begin{array}{lll}
\mathit{Distrib} & [\{u_1,\ldots,u_n\}](v) & \Longrightarrow \quad \{[u_1](v),\ldots,[u_n](v)\} \\
\mathit{Batch} & [v](\{u_1,\ldots,u_n\}) & \Longrightarrow \quad \{[v](u_1),\ldots,[v](u_n)\} \\
\mathit{Switch}_L & \{u_1,\ldots,u_n\} \to v & \Longrightarrow \quad \{u_1 \to v,\ldots,u_n \to v\} \\
\mathit{Switch}_R & u \to \{v_1,\ldots,v_n\} & \Longrightarrow \quad \{u \to v_1,\ldots,u \to v_n\} \\
\mathit{OpOnSet} & f(v_1,\ldots,\{u_1,\ldots,u_m\},\ldots,v_n) & \Longrightarrow \\
 & \{f(v_1,\ldots,u_1,\ldots,v_n),\ldots,f(v_1,\ldots,u_m,\ldots,v_n)\} &
\end{array}
$$

FIG. 4. The evaluation rules *Set* of the ρ_T-calculus

having the same left-hand side and use the ρ-term $a \to \{b, c\}$ which is reduced, by applying the evaluation rule Switch_R, to $\{a \to b, a \to c\}$. Thus, we can say that the ρ-term $[a \to \{b, c\}](a)$ describes the non-deterministic choice between the application of the rule $a \to b$ to the term a and the application of the rule $a \to c$ to the same term and this application is reduced to the set containing the results of the two applications, *i.e.* $\{\{b\}, \{c\}\}$.

Let us consider the ρ-term $[f(a \to b)](f(a))$ which is reduced, by using the rules *Cong* and *Fire*, to $\{f(\{b\})\}$ and then, by using the rule *OpOnSet* to $\{\{f(b)\}\}$. The two set symbols corresponding to the two applications of the evaluation rules *Fire* and *Cong* are thus preserved by the application of the rule *OpOnSet*.

A result of the form $\{\}$ (*i.e.* $\emptyset$) represents the failure of a rule application and such failures are *strictly* propagated in ρ-terms by the *Set* rules. For instance, the ρ-term $g([a \to b](c), \{a\})$ is reduced to $g(\emptyset, \{a\})$ and then, by using the rule *OpOnSet*, to $\emptyset$. One should notice that in this case, the information on the number of *Fire* and *Congruence* rules used in the reduction of the sub-term $\{a\}$ is lost.

The rewrite relation generated by the evaluation rules *Fire*, *Congruence* and the *Set* rules is finer (*i.e.* contains more elements) than the standard one (without sets) and is obviously non-confluent. A reason for the non-confluence is the lack of a similar evaluation rule for the propagation of sets on sets.

2.4.4 Flattening sets in the ρ_T-calculus

We usually care about the set of results obtained by reducing the redexes and not about the exact trace of the reduction leading to these results. In what follows we present the way this behavior is described in the ρ-calculus.

We use the evaluation rule *Flat* in Figure 5 that flattens the sets and eliminates the (nested) set symbols. In this case, the information on the number of reduction steps is lost. Notice that this implies that failure (the empty set) is *not* strictly propagated on sets.

The same behavior can be described by two distinct evaluation rules: one that would just flatten the sets and thus preserve the number of set braces, and another

$$\mathit{Flat} \quad \{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\} \implies \{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\}$$

FIG. 5. The evaluation rules *Flat* of the ρ_T-calculus

one that would eliminate the nested set symbols.

This behavior of the calculus could be summarized by stating that failure propagation by the *Set* rules is strict on all operators but sets. We will see later that *Fire* may induce non-strict propagations in some particular cases (see Example 4.4 on page 26).

The design decision to use sets for representing reduction results has another important consequence concerning the handling of sets with respect to matching. Indeed, sets are just used to store results and we do not wish to make them part of the theory. We are thus assuming that the matching operation used in the *Fire* evaluation rule is *not* performed modulo the set axioms. As a consequence, this requires in some cases to use a strategy that pushes set braces outside the terms whenever possible.

Every time a ρ-term is reduced using the rules *Fire* and *Congruence* of the ρ_T-calculus, a set is generated. These evaluation rules are the ones that describe the application of a rewrite rule at the top level or deeper in a term. The set obtained when applying one of the above evaluation rules can trigger the application of the other evaluation rules of the calculus. These evaluation rules deal with the (propagation of) sets and compute a "set-normal form" for the ρ-terms by pushing out the set braces and flattening the sets.

Therefore, we consider that the evaluation rules of the ρ_T-calculus consist of a set of *deduction* rules (*Fire*, *Cong*, *CongFail*) and a set of *computation* rules (*Distrib*, *Batch*, Switch_L, Switch_R, *OpOnSet*, *Flat*) and that the reduction behaves as in deduction modulo [DHK98]. This means that we can consider the computation rules as describing a congruence modulo which the deduction rules are applied. In such an approach we say that $[f(a \to b)](f(a))$ reduces to $\{f(\{b\})\}$ which is equivalent to $\{f(b)\}$.

2.4.5 Using the ρ_T-calculus

The aim of this section is to make concrete the concepts we have just introduced by giving a few examples of ρ-terms and ρ-reductions. Many other examples could be found on the ELAN web page [Pro01].

The ρ_T-calculus using syntactic matching (*i.e.* an empty matching theory) is denoted $\rho_\emptyset$-calculus or simply ρ-calculus when there is no ambiguity. We denote by ρ_C-calculus, ρ_A-calculus and ρ_{AC}-calculus the ρ_T-calculus with a matching theory commutative, associative and associative-commutative respectively.

Simple functional programming Let us start with the functional part of the calculus and give the ρ-terms representing some λ-terms. For example, the λ-abstraction $\lambda x.(y\ x)$, where y is a variable, is represented as the ρ-rule $x \to [y](x)$. The application of the above term to a constant a, $(\lambda x.(y\ x)\ a)$ is represented in the ρ-calculus by the application $[x \to [y](x)](a)$. This application reduces, in the λ-calculus, to the term $(y\ a)$ while in the ρ-calculus the result of the reduction is the singleton $\{[y](a)\}$. When a functional representation $f(x)$ is chosen, the λ-term $\lambda x.f(x)$ is represented by the ρ-term $x \to f(x)$ and a similar result is obtained for its application. One should notice that for ρ-terms of this form (*i.e.* that have a variable as a left-hand side) the syntactic matching performed in the ρ-calculus is trivial, *i.e.* it never fails and gives only one result.

There is no difficulty to represent more elaborate λ-terms in the ρ-calculus. Let us consider the term $\lambda x.f(x)\ (\lambda y.y\ a)$ with the following β-derivation: $\lambda x.f(x)\ (\lambda y.y\ a) \longrightarrow_\beta \lambda x.f(x)\ a \longrightarrow_\beta f(a)$. The same derivation can be recovered in the ρ-calculus for the corresponding ρ-term: $[x \to f(x)]([y \to y](a)) \longrightarrow_{Fire} [x \to f(x)](\{a\}) \longrightarrow_{Batch} \{[x \to f(x)](a)\} \longrightarrow_{Fire} \{\{f(a)\}\} \longrightarrow_{Flat} \{f(a)\}$. Of course, several reduction strategies can be used in the λ-calculus and reproduced accordingly in the ρ-calculus. Indeed, we will see in Section 3.1 that the ρ-calculus strictly embeds the λ-calculus.

Rewriting Now, if we introduce contextual information in the left-hand sides of the ρ-rules we obtain classical rewrite rules as $f(a) \to f(b)$ or $f(x) \to g(x,x)$. When we apply such a rewrite rule, the matching can fail and consequently, the application of the rewrite rule can fail. As we have already insisted in the previous sections, the failure of a rewrite rule is not a meta-property in the ρ-calculus but is represented by an empty set (of results). For example, in standard term rewriting we say that the application of the rule $f(a) \to f(b)$ to the term $f(c)$ fails and therefore the term is unchanged. On the contrary, in the ρ-calculus the corresponding term $[f(a) \to f(b)](f(c))$ evaluates to $\emptyset$.

Since, in the ρ-calculus, there is no restriction on the rewrite rules construction, a rewrite rule may use a variable as left-hand side, as in $x \to x+1$, or it may introduce new variables, as in $f(x) \to g(x,y)$. The free variables of the rewrite rules from the ρ-calculus allow us to dynamically build classical rewrite rules. For example, in the application $[y \to (f(x) \to g(x,y))](a)$, the variable y is free in the rewrite rule $f(x) \to g(x,y)$ but bound in the rule $y \to (f(x) \to g(x,y))$. The above application is reduced to the set $\{f(x) \to g(x,a)\}$ containing a classical rewrite rule.

By using free variables in the right-hand side of a rewrite rule we can also "parameterize" the rules by "strategies", as in the term $y \to [f(x) \to [y](x)](f(a))$ where the term to be applied to x is not explicit in the rule $f(x) \to [y](x)$. When reducing the application $[y \to [f(x) \to [y](x)](f(a))](a \to b)$, the variable y from the rewrite rule is instantiated to $a \to b$ and thus, the result of the reduction is $\{b\}$.

Non-determinism When the matching is done modulo an equational theory we obtain interesting behaviors.

An associative matching theory allows us, for example, to express the fact that an expression can be parenthesized in different ways. Take, for example, the list operator $\circ$ that appends two lists with elements of a given sort *Elem*. Any object of sort *Elem* represents a list consisting of this only object. If we define the operator $\circ$ as associative, the rewrite rule describing the decomposition of a list can be written in the associative ρ_A-calculus $l \circ l' \to l$. When applying this rule to the list $a \circ b \circ c \circ d$ we obtain as result the ρ-term $\{a, a \circ b, a \circ b \circ c\}$. If the operator $\circ$ had not been defined as associative, we would have obtained as the result of the same rule application one of the singletons $\{a\}$ or $\{a \circ b\}$ or $\{a \circ (b \circ c)\}$ or $\{(a \circ b) \circ c\}$, depending on the way the term $a \circ b \circ c \circ d$ is parenthesized.

A commutative matching theory allows us, for example, to express the fact that the order of the arguments is not significant. Let us consider a commutative operator $\oplus$ and the rewrite rule $x \oplus y \to x$ that selects one of the elements of the tuple $x \oplus y$. In the commutative ρ_C-calculus, the application $[x \oplus y \to x](a \oplus b)$ evaluates to the set $\{a, b\}$ that represents the set of non-deterministic choices between the two results. In standard rewriting, the result is not well defined; should it be a or b?

We can also use an associative-commutative theory like, for example, when an operator describes multi-set formation. Let us go back to the $\circ$ operator, but this time we define it as associative-commutative and we use the rewrite rule $x \circ x \circ L \to L$ that eliminates doubleton from lists of sort *Elem*. Since the matching is done modulo associativity-commutativity, this rule eliminates the doubleton no matter what is their position in the structure built using the $\circ$ operator. For instance, in the ρ_{AC}-calculus the application $[x \circ x \circ L \to L](a \circ b \circ c \circ a \circ d)$ evaluates to $\{b \circ c \circ d\}$: the search for the two equal elements is done thanks to associativity and commutativity.

Another facility is due to the use of sets for handling non-determinism. This allows us to easily express the non-deterministic application of a set of rewrite rules to a term. Let us consider, for example, the operator $\otimes$ as a syntactic operator. If we want the same behavior as before for the selection of each element of the couple $x \otimes y$, two rewrite rules should be non-deterministically applied as in the following reduction: $[\{x \otimes y \to x, x \otimes y \to y\}](a \otimes b) \longrightarrow_{Distrib} \{[x \otimes y \to x](a \otimes b), [x \otimes y \to y](a \otimes b)\}$ $\longrightarrow_{Fire} \{\{a\}, \{b\}\} \longrightarrow_{Flat} \{a, b\}$.

2.5 Evaluation strategies for the ρ_T-calculus

The last component of a calculus, *i.e.* the strategy $\mathcal{S}$ guiding the application of its evaluation rules, is crucial for obtaining good properties for the ρ-calculus. For example, the main property analyzed for the ρ-calculus is confluence and we will see that if the rule *Fire* is applied under no conditions at any position of a ρ-term, confluence does not hold.

Let us now define formally the notion of strategy. We specialize here to the ρ-calculus, and the general definition can be found in [KKV95].

Definition 2.13
An *evaluation strategy* in the ρ-calculus is a subset of the set of all possible derivations.

For example, the $\mathcal{ALL}$ strategy is the set of *all* derivations, *i.e.* it imposes no restrictions. The empty strategy does not allow any reduction. Standard strategies are call by value or by name, leftmost innermost or outermost, lazy, needed.

The reasons for the non-confluence of the calculus are explained in Section 4 and a solution is proposed for obtaining a confluent calculus. The confluent strategy can be given explicitly or as a condition on the application of the rule *Fire*.

2.6 Summary

Starting from the notions introduced in the previous sections we give the definition of the ρ_T-calculus.

Definition 2.14
Given a set $\mathcal{F}$ of function symbols, a set $\mathcal{X}$ of variables, a theory T on $\varrho(\mathcal{F}, \mathcal{X})$ terms

having a decidable matching problem, we call ρ_T-calculus (or generically rewriting calculus) a calculus defined by:

1. a non-empty subset $\varrho_{-}(\mathcal{F}, \mathcal{X})$ of the $\varrho(\mathcal{F}, \mathcal{X})$ terms,
2. the (higher-order) substitution application to terms as defined in Section 2.2,
3. the theory T,
4. the set of evaluation rules $\mathcal{E}$: *Fire*, *Cong*, *CongFail*, *Distrib*, *Batch*, $Switch_L$, $Switch_R$, *OpOnSet*, *Flat*,
5. an evaluation strategy $\mathcal{S}$ that controls the application of the evaluation rules. The set $\varrho_{-}(\mathcal{F}, \mathcal{X})$ should be stable under the strategy controlled application of the evaluation rules.

We use the notation $\rho_T = (\varrho_{-}(\mathcal{F}, \mathcal{X}), T, \mathcal{S})$ to make apparent the main components of the rewriting calculus under consideration.

When the parameters of the general calculus are replaced with some specific values, different variants of the calculus are obtained. The remainder of this paper will be devoted, mainly, to the study of a specific instance of the ρ_T-calculus: the ρ-calculus.

2.7 Definition of the ρ-calculus

We define the ρ-calculus as the ρ_T-calculus where the matching theory T is restricted to first-order syntactic matching. As an instance of Definition 2.14 we get:

Definition 2.15
The ρ-calculus is the calculus defined by:

- the subset $\varrho_{\emptyset}(\mathcal{F}, \mathcal{X})$ of $\varrho(\mathcal{F}, \mathcal{X})$ whose rewrite rules are restricted to be of the form $u \to v$ where $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, *i.e.* u is a first-order term and thus does not contain any set, application or abstraction symbol,
- the higher-order substitution application to terms,
- the matching theory $T = \emptyset$, *i.e.* first-order syntactic matching,
- the set of evaluation rules $\mathcal{R}$ presented in Figure 6 (*i.e.* all the rules of the ρ-calculus but $Switch_L$),
- the evaluation strategy $\mathcal{ALL}$ that imposes no conditions on the application of the evaluation rules.

The ρ-calculus is therefore defined as the calculus $\rho_{\emptyset} = (\varrho_{\emptyset}(\mathcal{F}, \mathcal{X}), \emptyset, \mathcal{ALL})$.

Example 2.16
With the exception of the last term, all the ρ-terms from Example 2.3 are $\rho_{\emptyset}$-terms.

The following remarks should be made with respect to the restrictions introduced in the ρ-calculus:

- Since first-order syntactic matching is unitary (*i.e.* the match, when it exists, is unique) the meta-rule *Propagate* from Section 2.4 gives always as result either the

$Fire$	$[l \to r](t)$	$\Longrightarrow$	$\{\sigma r\}$ where $\{\sigma\} = \mathcal{S}olution(l \ll^{?}_{T} t)$
$Cong$	$[f(u_1,\ldots,u_n)](f(v_1,\ldots,v_n))$	$\Longrightarrow$	$\{f([u_1](v_1),\ldots,[u_n](v_n))\}$
$CongFail$	$[f(u_1,\ldots,u_n)](g(v_1,\ldots,v_m))$	$\Longrightarrow$	$\emptyset$
$Distrib$	$[\{u_1,\ldots,u_n\}](v)$	$\Longrightarrow$	$\{[u_1](v),\ldots,[u_n](v)\}$
$Batch$	$[v](\{u_1,\ldots,u_n\})$	$\Longrightarrow$	$\{[v](u_1),\ldots,[v](u_n)\}$
$Switch_R$	$u \to \{v_1,\ldots,v_n\}$	$\Longrightarrow$	$\{u \to v_1,\ldots,u \to v_n\}$
$OpOnSet$	$f(v_1,\ldots,\{u_1,\ldots,u_m\},\ldots,v_n)$ $\{f(v_1,\ldots,u_1,\ldots,v_n),\ldots,f(v_1,\ldots,u_m,\ldots,v_n)\}$	$\Longrightarrow$	
$Flat$	$\{u_1,\ldots,\{v_1,\ldots,v_n\},\ldots,u_m\}$	$\Longrightarrow$	$\{u_1,\ldots,v_1,\ldots,v_n,\ldots,u_m\}$

FIG. 6. The evaluation rules of the ρ-calculus

singleton $\{\sigma r\}$ or the empty set. Hence, the evaluation rule $Fire$ can be replaced by the following simpler two rules:

$$\begin{array}{llll} Fire' & [l \to r](\sigma l) & \Longrightarrow & \{\sigma r\} \\ Fire'' & [l \to r](t) & \Longrightarrow & \emptyset \\ & & & \text{if there exists no } \sigma \text{ s.t. } \sigma l = t \end{array}$$

- The evaluation rule $Switch_L$ can never be used in the ρ-calculus due to the restricted syntax imposed on $\rho_\emptyset$-terms.
- For a specific instance of the ρ_T-calculus, there is a strong relationship between the terms allowed on the left-hand side of the rule and the theory T. Intuitively, the theory T should be powerful enough to fire rule applications in a way consistent with the intended rewriting. For instance, it seems more interesting to use higher-order matching instead of syntactic or equational matching when the left-hand sides of rules contain abstractions and applications. This explains the restriction imposed in the ρ-calculus for the formation of left-hand sides of rules.
- The term restrictions are made only on the left-hand sides of rewrite rules and not on the right-hand side and this clearly leads to more terms than in λ-calculus or in term rewriting.
- The ρ-calculus is not terminating as $[\omega](\omega)$ is a ρ-term (see Example 2.3).

The case of decidable finitary equational theories will induce more technicalities but is conceptually similar to the case of the empty theory. The case of theories with infinitary or undecidable matching problems could be treated using constraint ρ-terms in the spirit of [KKR90], and will be studied in forthcoming works.

3 Encoding λ-calculus and term rewriting in the ρ-calculus

The aim of this section is to show in detail how the ρ-calculus can be used to give a natural encoding of the λ-calculus and term rewriting.

3.1 Encoding the λ-calculus

We briefly present some of the notions used in the λ-calculus, such as β-redex and β-reduction, that will be used in this part of the paper. The reader should refer to [HS86] and [Bar84] for a detailed presentation.

Let $\mathcal{X}$ be a set of variables, written x, y, etc. The terms of the λ-calculus are inductively defined by:

$$a ::= x \mid (a\ a) \mid \lambda x.a$$

Definition 3.1
The β-reduction is defined by the rule:

$$\mathit{Beta} \quad (\lambda x.M\ N) \quad \rightsquigarrow \quad \{x/N\}M$$

Any term of the form $(\lambda x.M)N$ is called a *β-redex*, and the term $\{x/N\}M$ is traditionally called its *contractum*. If a term P contains a redex, P can be *β-contracted* into P' which is denoted:

$$P \ \longrightarrow_\beta \ P'.$$

If Q is obtained from P by a finite (possibly empty) number of β-contractions we say that P *β-reduces* to Q and we denote:

$$P \ \overset{*}{\longrightarrow}_\beta \ Q.$$

Let us consider a restriction of the set of ρ-terms, denoted $\mathcal{F}_\lambda$, and inductively defined as follows:

$$\rho_\lambda\text{-terms} \quad t \quad ::= \quad x \mid \{t\} \mid [t](t) \mid x \rightarrow t$$

where $x \in \mathcal{X}$.

Definition 3.2
The ρ_λ-calculus is the ρ-calculus defined by:

- the $\mathcal{F}_\lambda$ terms,
- the higher-order substitution application to terms,
- the (matching) theory $T = \emptyset$,
- the set of evaluation rules of the ρ-calculus,
- the evaluation strategy $\mathcal{ALL}$ that imposes no conditions on the application of the evaluation rules.

Compared to the syntax of the general ρ-calculus, the rewrite rules allowed in the ρ_λ-calculus can only have a variable as left-hand side. Additionally, all the sets are singletons, hence one could consider an encoding not using sets. For uniformity purposes, we chose to stick to the same encoding approach.

$$\begin{array}{llcl}
Fire_{\lambda} & [x \to r](t) & \Longrightarrow & \{\{x/t\}r\ \} \\
Distrib_{\lambda} & [\{u\}](v) & \Longrightarrow & \{[u](v)\} \\
Batch_{\lambda} & [v](\{u\}) & \Longrightarrow & \{[v](u)\} \\
Switch_{\lambda} & x \to \{v\} & \Longrightarrow & \{x \to v\} \\
Flat_{\lambda} & \{\{v\}\} & \Longrightarrow & \{v\}
\end{array}$$

Fig. 7. The evaluation rules of the ρ_{λ}-calculus

Because of the syntactic restrictions we have just imposed, the evaluation rules of the ρ-calculus specialize to the ones described in Figure 7.

The evaluation rule $Fire_{\lambda}$ initiates in the ρ-calculus (as the β-rule in the λ-calculus) the application of a substitution to a term. The rules *Congruence* are not used and the rules *Set* and *Flat* can be specialized to singletons and describe how to push out the set braces.

An immediate consequence of the restricted syntax of the ρ_{λ}-calculus is that the matching performed in the evaluation rule $Fire_{\lambda}$ always succeeds and the solution of the matching equation that is necessarily of the form $x \ll^{?}_{\emptyset} t$ is always the singleton $\{\{x/t\}\}$.

At this moment we can notice that any λ-term can be represented by a ρ-term. The function φ that transforms terms in the syntax of the λ-calculus into the syntax of the ρ_{λ}-calculus is defined by the following transformation rules:

$$\begin{array}{lcl}
\varphi(x) & = & x, \text{ if } x \text{ is a variable} \\
\varphi(\lambda x.t) & = & x \to \varphi(t) \\
\varphi(t\ u) & = & [\varphi(t)](\varphi(u))
\end{array}$$

A similar translation function can be used in order to transform terms in the syntax of the ρ_{λ}-calculus into the syntax of the λ-calculus:

$$\begin{array}{lcl}
\delta(x) & = & x, \text{ if } x \text{ is a variable} \\
\delta(\{t\}) & = & \delta(t) \\
\delta([t](u)) & = & (\delta(t)\ \delta(u)) \\
\delta(x \to t) & = & \lambda x.\delta(t)
\end{array}$$

The reductions in the λ-calculus and in the ρ_{λ}-calculus are equivalent modulo the notations for the application and the abstraction and the handling of sets:

Proposition 3.3
Given two λ-terms t and t', if $t \longrightarrow_{\beta} t'$ then $\varphi(t) \overset{*}{\longrightarrow}_{\rho_{\lambda}} \{\varphi(t')\}$.
Given two ρ_{λ}-terms u and u', if $u \longrightarrow_{\rho_{\lambda}} u'$ then $\delta(u) \overset{*}{\longrightarrow}_{\beta} \delta(u')$.

Proof. We use an induction on $\longrightarrow_{\beta}$ and $\longrightarrow_{\rho_{\lambda}}$ respectively:

- If t is a variable x, then $t' = x$ and $\varphi(t) = \varphi(t') = x$.
- If $t = \lambda x.u$ then $t' = \lambda x.u'$ with $u \longrightarrow_{\beta} u'$ and we have $\varphi(t) = x \to \varphi(u)$. By induction, we have $\varphi(u) \overset{*}{\longrightarrow}_{\rho_{\lambda}} \{\varphi(u')\}$, and thus

$$\varphi(t) = x \to \varphi(u) \quad \overset{*}{\longrightarrow}_{\rho_{\lambda}} \quad x \to \{\varphi(u')\} \quad \longrightarrow_{Switch_{\lambda}} \quad \{x \to \varphi(u')\} = \{\varphi(t')\}$$

- If $t = (u\ v)$ then we have either $t' = (u'\ v)$ with $u \longrightarrow_\beta u'$, or $t' = (u\ v')$ with $v \longrightarrow_\beta v'$, or $t = \lambda x.u\ v$ and $t' = \{x/v\}u$.
  In the first case, we apply induction and we obtain
  $$\varphi(t) = [\varphi(u)](\varphi(v)) \overset{*}{\longrightarrow}_{\rho_\lambda} [\{\varphi(u')\}](\varphi(v)) \longrightarrow_{Distrib_\lambda} \{[\varphi(u')](\varphi(v))\} = \{\varphi(t')\}.$$
  The second case is similar,
  $$\varphi(t) = [\varphi(u)](\varphi(v)) \overset{*}{\longrightarrow}_{\rho_\lambda} [\{\varphi(u)\}](\varphi(v')) \longrightarrow_{Distrib_\lambda} \{[\varphi(u)](\varphi(v'))\} = \{\varphi(t')\}.$$
  In the third case $\varphi(t) = [x \to \varphi(u)](\varphi(v))$ and
  $$\varphi(t) = [x \to \varphi(u)](\varphi(v)) \longrightarrow_{Fire_\lambda} \{\{x/\varphi(v)\}\varphi(u)\} = \varphi(\{x/v\}u) = \varphi(t').$$
  Since the application of a substitution is the same in the λ-calculus and the ρ-calculus, we have, due to the definition of φ, $\varphi(\{x/v\}u) = \{x/\varphi(v)\}\varphi(u)$ and thus, the property is verified.

Since in the ρ_λ-calculus we can have only singletons and the δ transformation strips off the set symbols, the application of the evaluation rules $Distrib_\lambda$, $Batch_\lambda$, $Switch_\lambda$ and $Flat_\lambda$ corresponds to the identity in the λ-calculus.

- If $t = [\{u\}](v)$ then we have $t \longrightarrow_{Distrib_\lambda} \{[u](v)\}$. Since $\delta([\{u\}](v)) = \delta(u)\ \delta(v)$ and $\delta(\{[u](v)\}) = \delta(u)\ \delta(v)$, the property is verified.
- If $t = [x \to u](v)$ then $t \longrightarrow_{Fire_\lambda} \{\{x/v\}u\}$. We have
  $$\delta(t) = \lambda x.\delta(u)\ \delta(v) \longrightarrow_\beta \{x/\delta(v)\}\delta(u)\} = \delta(\{x/v\}u) = \delta(t').$$

The other cases are very similar to the first one and to their correspondents from the first part.

□

Example 3.4
We consider the three combinators $I = \lambda x.x$, $K = \lambda xy.x$ and $S = \lambda xyz.xz(yz)$ and their representation in the ρ-calculus:

- $I = x \to x$,
- $K = x \to (y \to x)$,
- $S = x \to (y \to (z \to [[x](z)]([y](z))))$.

and, as expected, to a reduction $SKK \overset{*}{\longrightarrow}_\beta I$ in the λ-calculus it corresponds the ρ_λ-reduction $[[S](K)](K) \overset{*}{\longrightarrow}_{\rho_\lambda} \{I\}$.

$$\begin{aligned}
[[S](K)](K) = \ & [[x \to (y \to (z \to [[x](z)]([y](z))))](x \to (y \to x))](x \to (y \to x)) \longrightarrow_{\rho_\lambda} \\
& [\{y \to (z \to [[x \to (y \to x)](z)]([y](z)))\}](x \to (y \to x)) \longrightarrow_{\rho_\lambda} \\
& \{[y \to (z \to [[x \to (y \to x)](z)]([y](z)))](x \to (y \to x))\} \longrightarrow_{\rho_\lambda} \\
& \{[y \to (z \to [\{y \to z\}]([y](z)))](x \to (y \to x))\} \longrightarrow_{\rho_\lambda} \\
& \{\{[y \to (z \to [y \to z]([y](z)))](x \to (y \to x))\}\} \longrightarrow_{\rho_\lambda} \\
& \{\{[y \to (z \to \{z\})](x \to (y \to x))\}\} \longrightarrow_{\rho_\lambda} \\
& \{\{\{[y \to (z \to z)](x \to (y \to x))\}\}\} \longrightarrow_{\rho_\lambda} \\
& \{\{\{\{z \to z\}\}\}\} \longrightarrow_{\rho_\lambda} \\
& \{z \to z\} = \{I\}
\end{aligned}$$

The need for adding a set symbol comes from the fact that in the ρ-calculus we are mainly interested in the application of terms to some other terms. From this point of view, the application of a term t to another term u reduces to the same thing as the application of the term $\{t\}$ to the same term u.

In the ρ_λ-calculus, we could have introduced an evaluation rule eliminating all set symbols. But as soon as failure, represented by the empty set, and non-determinism, represented by sets with more than one element, are introduced such an evaluation rule will not be meaningful anymore.

The confluence of the λ-calculus holds for any complete reduction strategy (*i.e.* a strategy that does not leave any redex un-reduced) and we would expect the same result for its ρ-representation. As we have already noticed, since in the ρ_λ-calculus all the rewrite rules are left-linear and all the sets are singletons, the confluence conditions that will be presented in Section 4.2 are always satisfied. Therefore, the evaluation rule $Fire_\lambda$ can be used on any ρ_λ-application without losing the confluence of the ρ_λ-calculus.

Proposition 3.5
The ρ_λ-calculus is confluent.

Notice finally that using the same technique, the λ-calculus with patterns of [PJ87] can be encoded as a sub-calculus of the ρ-calculus.

3.2 Encoding finite rewrite sequences

As far as it concerns term rewriting, we just recall the basic notions that are consistent with [DJ90, BN98] to which the reader is referred for a more detailed presentation.

A *rewrite theory* is a 4-tuple $\mathcal{R} = (\mathcal{X}, \mathcal{F}, E, R)$ where $\mathcal{X}$ is a given countably infinite set of variables, $\mathcal{F}$ a set of ranked function symbols, E a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities, and R a set of rewrite rules of the form $l \to r$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ satisfying $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.

In what follows we consider $E = \emptyset$ but we conjecture that all the results concerning the encoding of rewriting in ρ-calculus can be smoothly extended to any equational theory E.

Since the rewrite rules are trivially ρ-terms, the representation of rewrite sequences in the ρ-calculus is quite simple. We consider a restriction of the ρ-calculus where the right-hand sides of rewrite rules are terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. The rewrite rules are trivially translated in the ρ-calculus and the application of a rewrite rule at the top position of a term is represented using the ρ-operator $[\_](\_)$.

We want to show that for any derivation in a rewriting theory, a corresponding reduction can be found in the ρ-calculus. If we consider that a sub-term w of a term t is reduced to w' by applying some rewrite rule $(l \to r)$ and thus,

$$t_{\lceil w \rceil_p} \longrightarrow_{\mathcal{R}} t_{\lceil w' \rceil_p}$$

then, we can build immediately the ρ-term $t_{\lceil [l \to r](w) \rceil_p}$ with the reduction:

$$t_{\lceil [l \to r](w) \rceil_p} \longrightarrow_\rho t_{\lceil \{w'\} \rceil_p} \xrightarrow{*}_\rho \{t_{\lceil w' \rceil_p}\}.$$

The above construction method for the ρ-term with a ρ-reduction similar to that of the term t according to the rule $l \to r$ is very easy but allows us to find the correspondence for only one rewrite step. It is not easy to extend this representation for an unspecified number of reduction steps w.r.t. a set of rewrite rules and a systematic method for the construction of the corresponding ρ-term is desirable.

Proposition 3.6
Given a rewriting theory $\mathcal{T}_{\mathcal{R}}$ and two first order ground terms $t, t' \in \mathcal{T}(\mathcal{F})$ such that $t \xrightarrow{*}_{\mathcal{R}} t'$. Then, there exist the ρ-terms $u_1, \ldots, u_n$ built using the rewrite rules in $\mathcal{R}$ and the intermediate steps in the derivation $t \xrightarrow{*}_{\mathcal{R}} t'$ such that we have $[u_n](\ldots [u_1](t) \ldots) \xrightarrow{*}_{\rho_\emptyset} \{t'\}$.

Proof. We use induction on the length of the derivation $t \xrightarrow{*}_{\mathcal{R}} t'$.

The base case: $t \xrightarrow{0}_{\mathcal{R}} t$ (derivation in 0 steps)

We have immediately $[x \to x](t) \xrightarrow{0}_{\rho_\emptyset} \{t\}$.

Induction: $t \xrightarrow{n}_{\mathcal{R}} t'$ (derivation in n steps)

We consider that the rewrite rule $l \to r$ is applied at position p of the term $t'{\lceil w \rceil}_p$ obtained after $n-1$ reduction steps,

$$t \xrightarrow{n-1}_{\mathcal{R}} t'{\lceil w \rceil}_p \longrightarrow_{l \to r, p} t'{\lceil \theta r \rceil}_p$$

where θ is the grafting such that $\theta l = w$.

By induction, there exist the ρ-terms $u_1, \ldots, u_{n-1}$ such that we have the reduction $[u_{n-1}](\ldots [u_1](t) \ldots) \xrightarrow{*}_{\rho_\emptyset} \{t'{\lceil w \rceil}_p\}$. We consider the ρ-term $u_n = t'{\lceil l \to r \rceil}_p$ and we obtain the reduction
$[u_n](\ldots [u_1](t) \ldots) \xrightarrow{*}_{\rho_\emptyset} [t'{\lceil l \to r \rceil}_p](\{t'{\lceil w \rceil}_p\}) \longrightarrow_{Batch} \{[t'{\lceil l \to r \rceil}_p](t'{\lceil w \rceil}_p)\}$
$\xrightarrow{*}_{Congruence} \{\{t'{\lceil [l \to r](w) \rceil}_p\}\} \longrightarrow_{Fire} \{\{t'{\lceil \{\theta' r\} \rceil}_p\}\} \xrightarrow{*}_{OpOnSet} \{\{\{t'{\lceil \theta' r \rceil}_p\}\}\}$
$\xrightarrow{*}_{Flat} \{t'{\lceil \theta' r \rceil}_p\}$
where the substitution θ' is such that $\{\theta'\} = \mathcal{S}olution(l \ll^?_\emptyset w)$.

Since $\theta = \theta'$ and in this case substitution and grafting are identical, we obtain $t'{\lceil \theta' r \rceil}_p = t'{\lceil \theta r \rceil}_p$.

□

Until now we have used the evaluation rule *Cong* for constructing the reduction

$$[t^n{\lceil l_n \to r_n \rceil}_{p_n}](\ldots [t^2{\lceil l_2 \to r_2 \rceil}_{p_2}]([t^1{\lceil l_1 \to r_1 \rceil}_{p_1}](t)) \ldots) \xrightarrow{*}_{\rho} \{t'\}$$

that corresponds, in the ρ-calculus, to the reduction, in the rewrite theory,

$$t = t^1{\lceil w_1 \rceil}_{p_1} \longrightarrow_{l_1 \to r_1, p_1} t^2{\lceil w_2 \rceil}_{p_2} \longrightarrow_{l_2 \to r_2, p_2} \cdots \longrightarrow_{l_n \to r_n, p_n} t^n{\lceil w_n \rceil}_{p_n} = t'$$

As explained in Section 2.4, to any reduction performed using the rule *Cong* corresponds a reduction that is done using the rule *Fire*. Starting from the term u corresponding to a reduction in n (*Cong*) steps we build the term u' that reduces to the same term as u but using *Fire* reductions:

$$[t^n{\lceil l_n \rceil}_{p_n} \to t^n{\lceil r_n \rceil}_{p_n}](\ldots ([t^1{\lceil l_1 \rceil}_{p_1} \to t^1{\lceil r_1 \rceil}_{p_1}](t)) \ldots) \xrightarrow{*}_{\rho} \{t'\}$$

Remark 3.7
One can notice that the terms u_i used in the proof above are similar to the proof terms used in labeled rewriting logic [Mes92]. Indeed we can see the ρ-terms as a generalization of such proof terms where the ";" is used as a notation for the composition of terms, *i.e.* $[u]([v](t))$ is denoted $[v; u](t)$.

4 The confluence of ρ-calculus

It is easy to see, and we provide typical examples just below, that the ρ-calculus is non-confluent. The main reason for the confluence failure comes from the introduction in the syntax of the new function symbols for denoting sets, abstraction and application. It results in a conflict between the use of syntactic matching and the set representation for the reductions results. This leads, on one hand, to undesirable matching failures due to terms that are not completely evaluated or not instantiated. On the other hand, we can have sets with more than one element that can lead to undesirable results in a non-linear context or empty sets that are not strictly propagated. In this section, we summarize the results of [Cir00] to which the reader is referred for full details. In particular we show on typical examples the confluence problems and we give a sufficient condition on the evaluation strategy of the ρ-calculus that allows to restore confluence.

4.1 The raw ρ-calculus is not confluent

Let us begin to show typical examples of confluence failure. A first such situation occurs when reducing a (sub-)term of the form $u = [l \to r](t)$ by matching l and t and when either t contains a redex, or u is redex.

In Example 4.1.a the non-confluence is obtained when a matching failure results from a non-reduced sub-term of t but succeeds when the sub-term is reduced. A similar situation is obtained when the evaluation rule *Fire* gives the $\emptyset$ result due to a matching failure but the application of another evaluation rule before the rule *Fire* leads to a non-empty set as in Example 4.1.b.

Example 4.1

a.

$$\begin{tikzcd}
{[a \to b]([a \to a](a))} \arrow[d, "Fire(internal)"'] \arrow[dr, "Fire(external)"] & \\
{[a \to b](\{a\})} \arrow[d, "Batch"'] & \emptyset \\
\{[a \to b](a)\} \arrow[d, "Fire"'] & \\
\{\{b\}\} &
\end{tikzcd}$$

b.

$$\begin{tikzcd}
{[a \to a](\{a\})} \arrow[d, "Batch"'] \arrow[dr, "Fire"] & \\
\{[a \to a](a)\} \arrow[d, "Fire"'] & \emptyset \\
\{\{a\}\} &
\end{tikzcd}$$

In Example 4.2 one can notice that a term can be reduced to an empty set because of a matching failure implying its bound variables. The result can be different from the empty set if the reductions of the sub-terms containing the respective variables are carried out only after the instantiation of these variables.

Example 4.2

$$\begin{tikzcd}
& {[x \to [a \to b](x)](a)} \arrow[dl, "Fire"', "(external)"] \arrow[dr, "Fire", "(internal)"'] & \\
\{[a \to b](a)\} \arrow[d, "Fire"'] & & {[x \to \emptyset](a)} \arrow[d, "Switch_R"] \\
\{\{b\}\} & & \emptyset
\end{tikzcd}$$

In order to avoid this kind of situation we should prevent the reduction of an application $[l \to r](t)$ if the matching between the terms l and t fails due to the matching rules $VariableClash$ (Example 4.2) or $SymbolClash$ (Example 4.1.a, 4.1.b) and either some variables are not instantiated or some of the terms are not reduced, or the term t is a set.

The matching rules $VariableClash$ and $SymbolClash$ would be never applied if the set of functional positions of the term l was a subset of the set of functional positions of the term t. This is not the case in Example 4.2 where, in the term $[a \to b](x)$, a is a functional position and the corresponding position in the argument of the rewrite rule application is the variable position x. In Example 4.1.a and Example 4.1.b a functional position in the left-hand side of the rewrite rule corresponds to an abstraction and set position respectively and thus, the condition is not satisfied.

Therefore, we could consider that the evaluation rule $Fire$ is applied only when the condition on the functional positions is satisfied. Unfortunately, such a condition will not suffice for avoiding a non-appropriate matching failure due to the application of the rule $MergingClash$. As shown in Example 4.3, such a situation can be obtained if the left-hand side of the rewrite rule to be applied is not linear.

Example 4.3

$$\begin{tikzcd}
& {[g(x,x) \to x](g(a,[a \to a](a)))} \arrow[dl, "Fire"', "(internal)"] \arrow[dr, "Fire", "(external)"'] & \\
{[g(x,x) \to x](g(a,\{a\}))} \arrow[d, "OpOnSet\&Batch"'] \arrow[dr, "Fire", "(external)"'] & & \emptyset \\
\{[g(x,x) \to x](g(a,a))\} \arrow[d, "Fire"'] & \emptyset & \\
\{\{a\}\} & &
\end{tikzcd}$$

Another pathological case arises when the term t contains an empty set or a subterm that can be reduced to the empty set. Indeed, the application of the rule $Fire$ can lead to the non-propagation of the failure and thus, to non-confluence as in the next example:

Example 4.4

$$\begin{tikzcd}
& {[x \to b](\emptyset)} \arrow[dl, "Fire"'] \arrow[dr, "Batch"] & \\
\{b\} & & \emptyset
\end{tikzcd}$$

We mention that a rewrite rule is *quasi-regular* if the set of variables of the left-hand side is included in the set of variables of the right-hand side. In Section 4.2 we give a formal definition for the notion of quasi-regular rewrite rule that takes into consideration all the operators of the ρ-calculus. We have already seen in Example 4.4 that the non-propagation of the failure is obtained when non-quasi-regular rewrite rules are applied to a term containing $\emptyset$. When a quasi-regular rewrite rule is applied to a term containing $\emptyset$, the empty set is present in the term resulting from the application of a substitution of the form $\{x/\emptyset\}$ to the right-hand side of the rewrite rule (unlike in Example 4.4) and thus, the appropriate propagation of the $\emptyset$ is guaranteed.

Another nasty situation, well known, in particular in graph rewriting, is obtained due to uncontrolled copies of terms. When applying a non-right-linear rewrite rule to a term that contains sets with more than one element, or terms that can be reduced to such sets, we obtain undesirable results as in Example 4.5.

Example 4.5

$$\begin{tikzcd}
{[x \to g(x,x)](\{a,b\})} \arrow[d, "Fire"'] \arrow[dr, "Batch"] & \\
\{g(\{a,b\},\{a,b\})\} \arrow[d, "OpOnSet"'] & \{[x \to g(x,x)](a), [x \to g(x,x)](b)\} \arrow[d, "Fire"] \\
\{\{g(a,\{a,b\}), g(b,\{a,b\})\}\} \arrow[d, "OpOnSet"'] & \{\{g(a,a)\},\{g(b,b)\}\} \arrow[d, "Flat"] \\
\{\{\{g(a,a),g(a,b)\},\{g(b,a),g(b,b)\}\}\} \arrow[d, "Flat"'] & \{g(a,a),g(b,b)\} \\
\{g(a,a),g(a,b),g(b,a),g(b,b)\} &
\end{tikzcd}$$

To sum-up, the non-confluence is due to the application of the evaluation rule *Fire* too early in a derivation and the typical situations that we want to avoid consist in using the rule *Fire* for reducing an application:

- containing non-instantiated variables,
- containing non-reduced terms,
- containing a non-left-linear rewrite rule,
- of a non-right-linear rewrite rule to a term containing sets with more than one element,
- of a non-quasi-regular rewrite rule to a term containing empty sets.

We can notice that if we assume the computation rules (see Section 2.4) to be applied eagerly, then some, but unfortunately not all of the above confluence problems vanish. In particular, non-confluence examples involving sets, as Example 4.4 and Example 4.5, are overcome by an eager application of the computation rules.

4.2 Enforcing confluence using strategies

As we have just seen in the previous section, the possibility of having empty sets or sets with more than one element leads immediately to non-confluent reductions implying the evaluation rules *Fire* and *Congruence*. But the confluence could be restored under an appropriate evaluation strategy and, in particular, this strategy should guarantee a strict failure propagation and an appropriate handling of the sets with more than one element.

A first possible approach consists in reducing a ρ-term by initially applying all the rules handling the sets (*Distrib*, *Batch*, $Switch_L$, $Switch_R$, *OpOnSet*, *Flat*), *i.e.* the computation rules, and only when none of these rules can be applied, apply one of the rules *Fire*, *Cong*, *CongFail*, *i.e.* the deduction rules, to the terms containing no sets.

But an application can be reduced, by using the rule *Fire*, to an empty set or to a set containing several elements and thus, this strategy can still lead, as previously, to non-confluent reductions. Another disadvantage of this approach is that for no restriction of the ρ-calculus the proposed strategy is reduced to the trivial strategy $\mathcal{ALL}$.

Since the sets (empty or having more than one element) are the main cause of the non-confluence of the calculus, a natural strategy consists in reducing the application of a rewrite rule by respecting the following steps: instantiate and reduce the argument of the application, push out the resulting set braces by distributing them in the terms and only when none of the previous reductions is possible, use the evaluation rule *Fire*. We can easily express this strategy by imposing a simple condition for the application of the evaluation rule *Fire*.

Definition 4.6
We call *ConfStratStrict* the strategy which consists in applying the evaluation rule *Fire* to a redex $[l \to r](t)$ only if the term t is a first order ground term.

Proposition 4.7
When using the evaluation strategy *ConfStratStrict*, the ρ-calculus is confluent.

Proof. We consider the parallelization of the relation induced by the evaluation rules *Fire* and *Congruence* on one hand and the relation induced by the other rules of the calculus on the other hand. We show the confluence of the two relations and then use Yokouchi's Lemma [YH90] to prove the strong confluence of the relation obtained by combining the former relations. This latter relation is the transitive closure of the relation induced by the evaluation rules *Fire* and *Congruence*, and the evaluation rules handling sets.

The Yokouchi Lemma can be easily proved due to the strict conditions on the application of the rule *Fire* and thus to the absence of interaction between the evaluation rules of the calculus. □

The strategy *ConfStratStrict* is quite restrictive and we would like to define a general strategy that becomes trivial (*i.e.* imposes no restriction) when restricted to some simpler calculi, as the λ-calculus.

A confluent strategy emerges from the above counterexamples and allows the application of the evaluation rule $Fire$ only if a possible failure in the matching is preserved by the subsequent ρ-reductions and if the argument of the application cannot be reduced to an empty set or to a set having more than one element. Such a generic strategy consists in applying the evaluation rule $Fire$ to a redex $[l \to r](t)$ only if:

- $t \in \mathcal{T}(\mathcal{F})$ is a first order ground term

or

- the term t is such that if the matching $l \ll^?_\emptyset t$ fails then, for all term t' obtained by instantiating or reducing t, the matching $l \ll^?_\emptyset t'$ fails, and
- the term t cannot be reduced to an empty set or to a set having more than one element.

If we consider an instance of the ρ-calculus such that all the sets are singletons and all the applications are of the form $[x \to u](v)$ then, all the above conditions are always satisfied. Hence, we can say that in this case the previous strategy is equivalent to the strategy $\mathcal{ALL}$, *i.e.* it imposes no restriction on the reductions. One can notice that the ρ_λ-terms satisfy the previous conditions and thus, such a strategy imposes no restrictions on the reductions of this instance of the ρ-calculus.

The conditions imposed for the generic strategy when the term t is not a first order ground term are clearly not appropriate for an implementation of the ρ-calculus and thus, we must define operational strategies guaranteeing the confluence of the calculus. These strategies will impose some decidable conditions that correspond to (and imply) the ones proposed above.

We introduce in what follows a more operational and more restrictive strategy definition guaranteeing the matching *"coherence"* by imposing structural conditions on the terms l and t involved in a matching problem $l \ll^?_\emptyset t$. In order to ensure the matching failure preservation by the ρ-reductions, the failure must be generated only by different first order symbols in the corresponding positions of the two terms l and t. This property is always verified if the two terms are first order terms but an additional condition must be imposed if the term t contains ρ-calculus specific operators, as the abstraction or the application.

Definition 4.8

A ρ-term l *weakly subsumes* a ρ-term t if

$$\forall p \in \mathcal{FP}os(l) \cap \mathcal{P}os(t) \Rightarrow t(p) \in \mathcal{F}$$

Thus, a ρ-term l *weakly subsumes* a ρ-term t if for any functional position of the term l, either this position is not a position of the term t, or it is a functional position of the term t.

Remark 4.9

If $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ weakly subsumes t then, for any non-functional position (*i.e.* the position of a variable, an application, an abstraction or a set) in t, the corresponding position in l, if it exists, is a variable position. Thus, if the top position of t is not a functional position, then l is a variable.

One can notice that if a first order term l subsumes t, then l weakly subsumes t.

Example 4.10
The term $h(a, y, c)$ weakly subsumes the term $g(b, [x \to x](c))$ and the term $f(a)$ weakly subsumes the term $g(b, [x \to x](c))$. The term $g(a, y)$ weakly subsumes the term $g(b, [x \to x](c))$ while the term $f(a)$ does not weakly subsumes $f([x \to x](c))$.

Definition 4.11
We call *ConfStrat* the strategy which consists in applying the evaluation rule $Fire$ to a redex $[l \to r](t)$ only if:

- $t \in \mathcal{T}(\mathcal{F})$ is a first order ground term

or

- the term $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is linear and l weakly subsumes t, and
- the term t contains no set with more than one element and no empty set, and
- for all sub-term $[u \to w](v)$ of t, u subsumes v, and
- the term t contains no sub-term of the form $[u](v)$ where u is not an abstraction.

One should notice that the conditions imposed by the strategy *ConfStrat* are decidable even if the term t is not a first order ground term. One can clearly decide if a term is of the form $[u](v)$ or $[u \to w](v)$ as well as the number of elements of a finite set. The condition that l weakly subsumes t is simply a condition on the symbols on the same positions of the two terms and since matching is syntactic, then the subsumption condition is also decidable. Consequently, all the conditions used in the strategy *ConfStrat* are decidable.

The condition forbidding sub-terms of t of the form $[u](v)$ if u is not a rewrite rule is imposed in order to prevent the application of the evaluation rule $CongFail$ leading to an empty set result. If one considers a version of the ρ-calculus without the evaluation rules *Congruence* then, this last condition is no longer necessary in the strategy *ConfStrat*. Hence, all the terms of the representation of the λ-calculus in the ρ-calculus trivially satisfy the above conditions and in this case the strategy *ConfStrat* is equivalent to the strategy $\mathcal{ALL}$.

Proposition 4.12
When using the evaluation strategy *ConfStrat*, the ρ-calculus is confluent.

Proof. Starting from the evaluation rule $Fire$ expressed as a conditional rule guarded by the conditions defined in the strategy *ConfStrat* we define the relation $FireCong$ induced by this latter rule and the *Congruence* rules. The other evaluation rules of the calculus induce a second relation called Set.

We denote by $\longrightarrow_F$ and $\longrightarrow_S$ respectively, the compatible (context) closures of these two relations, and by $\overset{*}{\longrightarrow}_S$ the reflexive and transitive closure of $\longrightarrow_S$.

We prove the confluence of the relation $\overset{*}{\longrightarrow}_S \longrightarrow_F \overset{*}{\longrightarrow}_S$ and we use an approach similar to the one followed in [CHL96] for proving the confluence of $\lambda_{\Uparrow}$.

Thus, we have to prove the strong confluence of the relation $\longrightarrow_F$, the confluence and termination of $\longrightarrow_S$ and the compatibility between the two relations (*i.e.* Yokouchi's Lemma.

Using a polynomial interpretation we show that $\longrightarrow_S$ terminates and by analyzing the induced critical pairs we obtain the local confluence and consequently, the confluence of this relation.

The relation $\longrightarrow_F$ is not strongly confluent but we define the parallel version of this relation in the style of *Tait & Martin-Löf*. We denote this relation by $\longrightarrow_{F_\|}$ and we show that is strongly confluent.

The Yokouchi Lemma is proved using the conditions imposed on the application of the rule *Fire*. We obtain thus the strong confluence of the relation $\xrightarrow{*}_S\longrightarrow_{F_\|}\xrightarrow{*}_S$ and since this latter relation is the transitive closure of the relation $\xrightarrow{*}_S\longrightarrow_F\xrightarrow{*}_S$ we deduce the confluence of the calculus.

The proof is presented in full detail in [Cir00]. □

The relatively restrictive conditions imposed in strategy *ConfStrat* can be relaxed at the price of the simplicity of the strategy. The conditions that we want to weaken concern on one hand, the number of elements of the sets and on the other hand, the form of the rewrite rules.

First, the absence of sets having more than one element is necessary in order to guarantee a good behavior for the non-right-linear rewrite rules. The *right-linearity* of a rewrite rule is defined as the linearity of the right-hand side w.r.t. the variables of the left-hand side. For example, $x \to g(x, y)$ is right-linear, but $x \to g(x, x)$ is not right-linear. Moreover, the right-linearity can be imposed only to the operators different from the set symbols ($\{\_\}$) and thus, the rewrite rule $x \to \{f(x), f(x)\}$ can be considered right-linear. Intuitively, we do not need to impose right-linearity for sets since, due to the evaluation rule *Flat*, they do not lead to non-convergent reductions as in Example 4.5.

Definition 4.13

The rewrite rule $l \to r$ is *hereditary right-linear* if any sub-term of r that is not a set is linear w.r.t. the free variables of l and any rewrite rule of r is hereditary right-linear.

The application of a rewrite rule which is not hereditary right-linear to a set with more than one element can lead to non-convergent reductions, as shown in Example 4.5, but this is not the case if the applied rewrite rule is hereditary right-linear:

Example 4.14

$$\begin{tikzcd}
{[x \to \{x, f(x)\}](\{a, b\})} \arrow[d, "Fire"'] \arrow[rd, "Batch"] & \\
\{\{a, b\}, f(\{a, b\})\} \arrow[d, "OpOnSet"'] & \{[x \to \{x, f(x)\}](a), [x \to \{x, f(x)\}](b)\} \arrow[d, "Fire"] \\
\{\{a, b\}, \{f(a), f(b)\}\} \arrow[rd, "Flat"'] & \{\{\{a, f(a)\}\}, \{\{b, f(b)\}\}\} \arrow[d, "Flat"] \\
& \{a, b, f(a), f(b)\}
\end{tikzcd}$$

On another hand, in order to guarantee the strict propagation of the failure, we impose that the evaluation rule *Fire* is applied only if the argument of the application is not an empty set and it cannot lead to an empty set. In Example 4.4 we can notice that the free variables of the left-hand side of the rewrite rule are not preserved in the right-hand side of the rule. If the rewrite rule $l \to r$ of the application preserves the variables of the left-hand side in the right-hand side (*e.g.* $x \to x$), the application of a substitution replacing one of these variables with an empty set (*e.g.* $\{x/\emptyset\}$) to r leads to a term containing $\emptyset$ and thus, which is possibly reduced to $\emptyset$.

We define thereafter more formally the rewrite rules preserving the variables and we present a new strategy defined using this property. First, we introduce a concept similar to that of free variable but, by considering this time the not-deterministic nature of the sets.

Definition 4.15
The set of *present variables* of a ρ-term t is denoted by $PV(t)$ and is defined by:

1. if $t = x$ then $PV(t) = \{x\}$,
2. if $t = \{u_1, \dots, u_n\}$ then $PV(t) = \bigcap_{i=1\dots n} PV(u_i)$, $(PV(\emptyset) = \mathcal{X})$,
3. if $t = f(u_1, \dots, u_n)$ then $PV(t) = \bigcup_{i=1\dots n} PV(u_i)$, $(PV(c) = \emptyset \ if \ c \in \mathcal{T}(\mathcal{F}))$,
4. if $t = [u](v)$ then $PV(t) = PV(u) \cup PV(v)$,
5. if $t = u \to v$ then $PV(t) = PV(v) \setminus FV(u)$.

The set of *free variables* of a set of ρ-terms is the union of the sets of free variables of each ρ-term while the set of *present variables* of a set of ρ-terms is the intersection of the sets of free variables of each ρ-term. We can say that a variable is *present* in a set only if it is present in all the elements of the set. For example, $PV(\{x, y, x\}) = \emptyset$ and $PV(\{x, g(x, y)\}) = \{x\}$.

Definition 4.16
We say that the ρ-rewrite rule $l \to r$ is quasi-regular if $FV(l) \subseteq PV(r)$ and any rewrite rule of r is quasi-regular.

Intuitively, to each free variable of the left-hand side of a quasi-regular rewrite rule corresponds, in a deterministic way, a free variable in the right-hand side of the rule. For any set ρ-term in the right-hand side, the correspondence with the free variables of the left-hand side should be verified for each element of the set.

Example 4.17
The rewrite rule $x \to g(x, y)$ is quasi-regular while the rewrite rule $x \to \{x, y\}$ is non-quasi-regular.

The rewrite rule $\{f(x), g(x, x)\} \to x$ is quasi-regular while $\{f(x), g(x, y)\} \to x$ is non-quasi-regular. If the definition of quasi-regular rewrite rules had asked for the condition $PV(l) \subseteq PV(t)$ instead, then the second rewrite rule would have become quasi-regular as well. This is not desirable since the rewrite rule $\{f(x), g(x, y)\} \to x$ reduces to $\{f(x) \to x, g(x, y) \to x\}$ and only the first one is quasi-regular.

In the particular case of the ρ-calculus, since the left-hand side of a rewrite rule $l \to r$ must be a first-order term (*i.e.* $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$), we have $FV(l) = PV(l) = \mathcal{V}ar(l)$ and thus the condition from Definition 4.16 can be changed to $\mathcal{V}ar(l) \subseteq PV(t)$.

Let us consider the application a quasi-regular rewrite rule $l \to r$ to a term t giving as result the term $\{\sigma r\}$, where σ is the matching substitution between l and t. If $\emptyset$ is a sub-term of t and if l weakly subsumes r, then $\emptyset$ is in σ. Since the rewrite rule is quasi-regular, we have $\mathcal{D}om(\sigma) \subseteq PV(r)$ and thus, we are sure that $\emptyset$ is a sub-term of σr. Furthermore, if $\emptyset$ instantiated a variable of a set in σr then it is present in all the elements of the set and thus, we avoid non-confluent results as the ones in Example 4.4.

Example 4.18
A quasi-regular rule applied to $\emptyset$ gives only one result:

$$\begin{tikzcd}
 & {[x \to \{x, g(x,a)\}](\emptyset)} \arrow[dl, "Switch_L"'] \arrow[dd, "Batch"'] \arrow[dr, "Fire"] & \\
{[\{x \to x, x \to g(x,a)\}](\emptyset)} \arrow[d, "Distrib"'] & & \{\emptyset, g(\emptyset, a)\} \arrow[d, "OpOnSet"] \\
\{[x \to x](\emptyset), [x \to g(x,a)](\emptyset)\} \arrow[r, "Fire\&OpOnSet", "\&Flat"'] & \emptyset & \{\emptyset, \emptyset\} \arrow[l, "Flat"']
\end{tikzcd}$$

while a non-quasi-regular one yields two different results as shown in Example 4.4.

One should notice that if a rewrite rule $l \to r$ is reduced by the evaluation rule *Switch_R* to a set of rewrite rules, each of these rules is quasi-regular and thus the strict propagation of the empty set is ensured on all the right-hand sides of the obtained rewrite rules.

Definition 4.19
We call *ConfStratLin* the strategy which consists in applying the evaluation rule *Fire* to a redex $[l \to r](t)$ only if $t \in \mathcal{T}(\mathcal{F})$ is a first order ground term *or*:

- the term $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is linear and l weakly subsumes t,

and

- *either*
  – $l \to r$ is quasi-regular
  or
  – the term t contains no empty set, and
  – for all sub-term $[u \to w](v)$ of t, u subsumes v, and
  – the term t contains no sub-term of the form $[u](v)$ where u is not an abstraction.

and

- *either*
  – $l \to r$ is hereditary right-linear
  or
  – the term t contains no set with more than one element.

Compared to the strategy *ConfStrat* we added the possibility to test either the quasi-regular condition on the rewrite rule $l \to r$ or the conditions on the reducibility of the term t to an empty set. Moreover, if the rewrite rule is hereditary right-linear we allow arguments containing sets having more than one element. Since one can clearly decide if a rule is quasi-regular or hereditary right-linear, all the conditions used in the strategy *ConfStratLin* are decidable.

Proposition 4.20
When using the evaluation strategy *ConfStratLin*, the ρ-calculus is confluent.

Proof. The same approach as for the strategy *ConfStrat* is used but some additional diagrams corresponding to the reductions that where not possible before are considered. These new cases are mainly introduced in the proof of Yokouchis's Lemma. The proof is detailled in [Cir00]. □

When using a calculus integrating reduction modulo an equational theory (*e.g.* associativity and commutativity), as explained in Section 2.4, the overall confluence proof is different but uses lemmas similar to the ones of the former case. Therefore, we conjecture that Proposition 4.12 and Proposition 4.20 can be extended to a ρ_E-calculus modulo a specific decidable and finitary equational matching theory E.

5 Conclusion

We have presented the ρ_T-calculus together with some of its variants obtained as instances of the general framework. By making explicit the notion of rule, rule application and application result, the ρ_T-calculus allows us to describe in a simple yet very powerful and uniform manner algebraic and higher-order capabilities. This provides therefore a simple and natural framework for their combination.

In the ρ_T-calculus the non-determinism is handled by using sets of results and the rule application failure is represented by the empty set. Handling sets is a delicate problem and we have seen that the raw ρ-calculus, where the evaluation rules are not guided by a strategy, is not confluent. When an appropriate but rather natural generalized call-by-value evaluation strategy is used, the calculus is confluent.

The ρ-calculus is both conceptually simple as well as quite expressive. This allows us to represent the terms and reductions from λ-calculus and rewriting. We conjecture that, following the lines of [Vir96], it is also simple to encode other calculi of interest like the π-calculus.

Part II, is devoted to the use of an extension of the calculus powerful enough to encode rewriting strategies, conditional rewriting and to give a semantics to the ELAN language. We refer to the conclusion of *Part II* for a presentation of the ongoing and future works on the ρ-calculus.

Acknowledgments

We would like to thank Hélène Kirchner, Pierre-Etienne Moreau and Christophe Ringeissen from the Protheo Team for the useful interactions we had on the topics of this paper, Vincent van Oostrom for suggestions and pointers to the literature, Roberto Bruni and David Wolfram for their detailed and very useful comments on a preliminary version of this work and Delia Kesner for fruitful discussions. We are grateful to Luigi Liquori for many comments and exciting discussions on the ρ-calculus and its applications. Many thanks also to Thérèse Hardin and Nachum Dershowitz for their interest, encouragements and helpful suggestions for improvement. Finally special thanks are due to the referees for the very complete and careful reading of the paper as well as constructive and useful remarks.

References

[AK92] M. Adi and C. Kirchner. Associative commutative matching based on the syntacticity of the AC theory. In F. Baader, J. Siekmann, and W. Snyder, editors, *Proceedings 6th International Workshop on Unification, Dagstuhl (Germany)*. Dagstuhl seminar, 1992.

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics.* Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam,

1984. Second edition.

[BKKR01] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[BT88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*, pages 82–90, 1988.

[CHL96] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cir00] H. Cirstea. *Calcul de réécriture : fondements et applications*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.

[CK99] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[Col88] L. Colson. Une structure de données pour le λ-calcul typé. Private Communication, 1988.

[DDHY92] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE computer society, 1992.

[Der85] N. Dershowitz. Computing with rewrite systems. *Information and Control*, 65(2/3):122–157, 1985.

[DHK95] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions, extended abstract. In D. Kozen, editor, *Proceedings of LICS'95*, pages 366–374, San Diego, June 1995.

[DHK98] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz`.

[DHKP96] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of JICSLP'96*, Bonn (Germany), September 1996. The MIT press.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Dow94] G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.

[Eke95] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[Eke96] S. Eker. Fast matching in combinations of regular equational theories. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[FH83] F. Fages and G. Huet. Unification and matching in equational theories. In *Proceedings Fifth Colloquium on Automata, Algebra and Programming, L'Aquila (Italy)*, volume 159 of *Lecture Notes in Computer Science*, pages 205–220. Springer-Verlag, 1983.

[GBT89] J. Gallier and V. Breazu-Tannen. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *16th Colloquium Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 1989.

[GKK$^+$87] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda-calculus.* Cambridge University, 1986.

[Hue73] G. Huet. A mechanization of type theory. In *Proceeding of the third international joint conference on artificial intelligence*, pages 139–146, 1973.

[Hue76] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω.* Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[JO97] J.-P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 28 February 1997.

[Kah87] G. Kahn. Natural semantics. Technical Report 601, INRIA Sophia-Antipolis, February 1987.

[Kes93] D. Kesner. *La définition de fonctions par cas à l'aide de motifs dans des langages applicatifs.* PhD thesis, Université de Paris XI, December 1993.

[KK99] C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKR90] C. Kirchner, H. Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[KKV95] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[Klo90] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[KR98] C. Kirchner and C. Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998.

[KvOvR93] J. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

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

[Mil84] R. Milner. A proposal for standard ML. In *Proceedings ACM Conference on LISP and Functional Programming*, 1984.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen, Germany, December 1989*, volume 475 of *Lecture Notes in Computer Science*, pages 253–281. Springer-Verlag, 1991.

[MuP96] MuPAD Group, Benno Fuchssteiner et al. *MuPAD User's Manual - MuPAD Version 1.2.2.* John Wiley and sons, Chichester, New York, first edition, march 1996. includes a CD for Apple Macintosh and UNIX.

[Nip89] T. Nipkow. Combining matching algorithms: The regular case. In N. Dershowitz, editor, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, volume 355 of *Lecture Notes in Computer Science*, pages 343–358. Springer-Verlag, April 1989.

[NP98] T. Nipkow and C. Prehofer. Higher-order rewriting and equational reasoning. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume I: Foundations.* Kluwer, 1998.

[O'D77] M. J. O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science.* Springer-Verlag, 1977.

[Oka89] M. Okada. Strong normalizability for the combined system of the typed λ calculus and an arbitrary convergent term rewrite system. In G. H. Gonnet, editor, *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation: ISSAC '89 / July 17–19, 1989, Portland, Oregon*, pages 357–363, New York, NY 10036, USA, 1989. ACM Press.

[Pad96] V. Padovani. *Filtrage d'ordre supérieur.* Thèse de Doctorat d'Université, Université Paris VII, 1996.

[Pad00] V. Padovani. Decidability of fourth-order matching. *Mathematical Structures in Computer Science*, 3(10):361–372, June 2000.

[Pag98] B. Pagano. X.R.S : Explicit Reduction Systems - A First-Order Calculus for Higher-Order Calculi. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, LNAI 1421, pages 72–87, Lindau, Germany, July 5–July 10, 1998. Springer-Verlag.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages.* Prentice Hall, Inc., 1987.

[Pro01] Protheo Team. The ELAN home page. WWW Page, 2001. `http://elan.loria.fr`.

[Rin96] C. Ringeissen. Combining Decision Algorithms for Matching in the Union of Disjoint Equational Theories. *Information and Computation*, 126(2):144–160, May 1996.

[vD96] A. van Deursen. An Overview of ASF+SDF. In *Language Prototyping*, pages 1–31. World Scientific, 1996. ISBN 981-02-2732-9.

[vdBvDK$^+$96] M. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *AMAST '96*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[Vir96] P. Viry. Input/Output for ELAN. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[Vit94] M. 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.

[vO90] V. van Oostrom. Lambda calculus with patterns. Technical report, Vrije Universiteit, Amsterdam, November 1990.

[Wol93] D. A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[Wol99] S. Wolfram. *The Mathematica Book*, chapter Patterns, Transformation Rules and Definitions. Cambridge University Press, 1999. ISBN 0-521-64314-7.

[YH90] H. Yokouchi and T. Hikita. A rewriting system for categorical combinators with multiple arguments. *SIAM Journal of Computing*, 19(1), February 1990.

Contents

Received