The rewriting calculus

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 is a new calculus that integrates in a uniform and simple setting first-order rewriting, λ-calculus and non-deterministic computations. This paper describes the calculus from its syntax to its basic properties in the untyped case. We show how it embeds first-order conditional rewriting and λ-calculus. Finally we use the ρ-calculus to give an operational semantics to the rewrite based language ELAN.

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 a 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, as well as their semantics.

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.

But the application of a rewrite rule may 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 we recover 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 $\lambda x.t$ could 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). 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 $\lambda(PAIR\ x\ y).x$ that selects the first element of a pair and the application $\lambda(PAIR\, x\, y).x\ (PAIR\, a\, b)$ that evaluates to a. The representation in the ρ-calculus of the first λ-term is $PAIR(x,y) \to x$ and the corresponding application $[PAIR(x,y) \to x](PAIR(a,b))$ 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 create a new redex.

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 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, BKK98] 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 the confluence and 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 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 show that for any reduction in the λ-calculus or conditional rewriting a corresponding natural reduction in the ρ-calculus can be found.

We extend the encoding of conditional rewriting in the ρ-calculus to more complicated rules like the conditional rewrite rules with local assignments from the ELAN language. The non-determinism that in ELAN is handled mainly by two basic strategy operators is represented in the ρ-calculus by means of sets. We show finally how the ρ-calculus provides a semantics to ELAN programs.

1.6 Structure of this paper

The purpose of this paper 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, possibly conditional, term rewriting. We also show, and indeed this was our first historical motivation, 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. Then, in Section 3, we restrict to the $\rho_\emptyset$-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 show in Section 4 how ρ-calculus can be used to encode in a uniform way term rewriting and λ-calculus. In Section 5 we extend the basic ρ-calculus with a new operator and define term traversal and fixed-point operators using the existing ρ-operators. The encoding of non-conditional and conditional term rewriting by using the ρ-operators defined in Section 5 is presented in Section 6. The calculus is finally used in Section 7 in order to give an operational semantics to the rules used in the ELAN language.

We conclude by providing some of the research directions that are of main interest in the development of this formalism and in the context of ELAN and more generally of rewrite based languages as ASF+SDF [Kli93], ML [Mil84], Maude [CELM96] or CafeOBJ [FN97].

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:

- 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.
- 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.
- The *matching algorithm* used to bind variables to their actual values. In the case of ρ_T-calculus, this is in general higher-order matching. But in practical cases it will be higher-order-pattern matching, or equational matching, or simply syntactic matching and their combination. 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.
- The *evaluation rules* describing the way the calculus operates. It is the glue between the previous components and the simplicity and clarity of these rules are fundamental for the calculus usability.
- 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 $t \to u$ is a ρ-term (abstraction or rewrite).

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

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

where $x \in \mathcal{X}$ and $f \in \mathcal{F}$.

A term may be viewed as a *finite labeled 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 $\mathcal{FP}os(t)$.

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 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. We consider that the symbols $\{\}$ and $\emptyset$ both represent the empty set. For the terms of the form $\{t_1, \ldots, t_n\}$ we assume, as usually, that the comma is associative, commutative and idempotent.

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 could 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 3 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 $\{a + y\}$.

- $[y \to [x \to x+y](b)]([x \to x](a))$; 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*

As 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 $t = \{u_1, \ldots, u_n\}$ then $FV(t) = \bigcup_{i=1,\ldots,n} FV(u_i)$,
3. if $t = f(u_1, \ldots, u_n)$ 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:

- $\alpha_{\mathcal{Y}}(x) = x$,
- $\alpha_{\mathcal{Y}}(\{t\}) = \{\alpha_{\mathcal{Y}}(t)\}$,
- $\alpha_{\mathcal{Y}}(f(u_1, \ldots, u_n)) = f(\alpha_{\mathcal{Y}}(u_1), \ldots, \alpha_{\mathcal{Y}}(u_n))$,
- $\alpha_{\mathcal{Y}}([t](u)) = [\alpha_{\mathcal{Y}}(t)](\alpha_{\mathcal{Y}}(u))$,
- $\alpha_{\mathcal{Y}}(u \to v) = \alpha_{\mathcal{Y}}(u) \to \alpha_{\mathcal{Y}}(v)$, 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 $(x, u) \in \theta$
- $\Theta(\{t_1, \dots, t_n\}) = \{\Theta(t_1), \dots, \Theta(t_n)\}$
- $\Theta(f(t_1 \dots t_n)) = f(\Theta(t_1) \dots \Theta(t_n))$
- $\Theta([t](u)) = [\Theta(t)](\Theta(u))$
- $\Theta(u \to v) = \Theta(u') \to \Theta(v')$

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

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:
$u' = \{y_i \mapsto z_i\}_{y_i \in FV(u)}\ \alpha_{FV(u) \cup \mathcal{V}ar(\theta)}(u)$,
$v' = \{y_i \mapsto z_i\}_{y_i \in FV(u)}\ \alpha_{FV(u) \cup \mathcal{V}ar(\theta)}(v)$.
The set of variables $\{x_1, \dots, x_n\}$ 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.

Recall that $\{x_1/t_1, \dots, x_n/t_n\}$ is the simultaneous substitution of the variables $x_1, \dots, x_n$ by the terms $t_1, \dots, t_n$ and not the composition $\{x_1/t_1\} \dots \{x_n/t_n\}$.
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 algorithm fails (*i.e.* returns $\mathbf{F}$) 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 primarily 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, Dow92, 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 the symbol $\wedge$ is assumed to be associative and commutative.

Decomposition	$(f(t_1,\dots,t_n) \ll^?_\emptyset f(t'_1,\dots,t'_n)) \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\bigwedge_{i=1\dots n} t_i \ll^?_\emptyset t'_i \wedge P$
SymbolClash	$(f(t_1,\dots,t_n) \ll^?_\emptyset g(t'_1,\dots,t'_m)) \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ if $f \neq g$
MergingClash	$(x \ll^?_\emptyset t) \wedge (x \ll^?_\emptyset t') \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ if $t \neq t'$
VariableClash	$(f(t_1,\dots,t_n) \ll^?_\emptyset x) \wedge P$	$\mapsto\!\!\!\!\rightarrow$	$\mathbf{F}$ 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

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

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 [Deu96]. 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 the 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*:

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

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.

$$Fire \quad [l \to r](t) \quad \Longrightarrow \quad r\langle\langle Solution(l \ll^?_T t)\rangle\rangle$$

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.* $Solution(l \ll^?_T t)$). 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.

We should point out that, as in λ-calculus an application can always be evaluated, but unlike in λ-calculus, the set of results could 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.

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 $f(u_1, \dots, u_n)$ (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.

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

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*. 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.

Distrib	$[\{u_1, \ldots, u_n\}](v)$	$\Longrightarrow$	$\{[u_1](v), \ldots, [u_n](v)\}$
Batch	$[v](\{u_1, \ldots, u_n\})$	$\Longrightarrow$	$\{[v](u_1), \ldots, [v](u_n)\}$
$Switch_L$	$\{u_1, \ldots, u_n\} \to v$	$\Longrightarrow$	$\{u_1 \to v, \ldots, u_n \to v\}$
$Switch_R$	$u \to \{v_1, \ldots, v_n\}$	$\Longrightarrow$	$\{u \to v_1, \ldots, u \to v_n\}$
OpOnSet	$f(v_1, \ldots, \{u_1, \ldots, u_m\}, \ldots, v_n)$	$\Longrightarrow$	$\{f(v_1, \ldots, u_1, \ldots, v_n), \ldots, f(v_1, \ldots, u_m, \ldots, v_n)\}$

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

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

$\{[a \to b](a), [a \to c](a)\}$). Moreover, we can factorize a set of rewrite rules 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. This section presents the way this behavior is described in the ρ-calculus.

An evaluation rule that would properly preserve the number of set braces, is the evaluation rule *FlatOne*:

$$FlatOne \quad \{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\} \implies \{\{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\}\} \quad \text{if } m \in \mathbf{N}^*$$

The drawback of this solution is the difference that we make between identical terms, but obtained after a different number of reduction steps. For example, using this approach, the term $\{\{a\}\}$ does not reduce to $\{a\}$ which is suitable in a calculus where we are interested in the result of the computation and not in the way it was obtained.

Because of this last reason, we can use the evaluation rule *Elim* that eliminates the (nested) set symbols:

$$Elim \quad \{\{u_1, \ldots, u_m\}\} \implies \{u_1, \ldots, u_m\}$$

Merging the two previous ideas leads to an approach where we directly flatten the sets and forget about the number of braces by using the evaluation rule *Flat* in Figure 5 that combines the rules *FlatOne* and *Elim*. In this case, the information on the number of reduction steps is lost. This latter approach is used in the ρ_T-calculus for flattening the sets. Notice that this implies that failure (the empty set) is *not* strictly propagated on sets.

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

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

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

Fire may induce non-strict propagations in some particular cases (see Example 3.6 on page 21).

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.

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 [Pro00].

The ρ_T-calculus using syntactic matching (*i.e.* an empty matching theory) is denoted by $\rho_\emptyset$-calculus. 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 $\rho_\emptyset$-calculus by the application $[x \to [y](x)](a)$. This application reduces, in the λ-calculus, to the term $(y\ a)$ while in the $\rho_\emptyset$-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 $\rho_\emptyset$-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 $\rho_{\emptyset}$-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 $\rho_{\emptyset}$-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 $\rho_{\emptyset}$-calculus. Indeed, we will see in Section 4.1 that the $\rho_{\emptyset}$-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 $\rho_{\emptyset}$-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 returned unchanged. On the contrary, in the $\rho_{\emptyset}$-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 $x \circ x \circ L \rightarrow L$ that eliminates doubletons from lists of sort *Elem*. Since the matching is done modulo associativity-commutativity, this rule eliminates the doubletons no matter what is their position in the structure built using the $\circ$ operator. For instance, in the ρ_{AC}-calculus the application $[x \circ x \circ L \rightarrow L](a \circ b \circ c \circ a \circ d)$ 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: $[\{x \otimes y \rightarrow x, x \otimes y \rightarrow y\}](a \otimes b) \longrightarrow_{Distrib} \{[x \otimes y \rightarrow x](a \otimes b), [x \otimes y \rightarrow y](a \otimes b)\}$ $\longrightarrow_{Fire} \{\{a\}, \{b\}\} \longrightarrow_{Flat} \{a, b\}$.

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, could be crucial for obtaining good properties for the ρ-calculus. For example, the main property analyzed for the ρ-calculus is the 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 $\mathcal{NONE}$ 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 3 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.

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 $\rho_\emptyset$-calculus.

3 The $\rho_\emptyset$-calculus

We now define the $\rho_\emptyset$-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 3.1
The $\rho_\emptyset$-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 $\mathcal{NONE}$ that imposes no conditions on the application of the evaluation rules.

The $\rho_\emptyset$-calculus is therefore defined as the calculus $\rho_\emptyset = (\varrho_\emptyset(\mathcal{F}, \mathcal{X}), \emptyset, \mathcal{NONE})$.

EXAMPLE 3.2
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 $\rho_\emptyset$-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 singleton $\{\sigma r\}$ or the empty set. Hence, the evaluation rule *Fire* can be replaced by the following simpler two rules:

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

- The evaluation rule $Switch_L$ can never be used in the $\rho_\emptyset$-calculus due to the restricted syntax imposed on $\rho_\emptyset$-terms.

Fire	$[l \to r](t)$	$\Longrightarrow$	$r\langle\langle Solution(l \ll^?_\emptyset t)\rangle\rangle$
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	$f(v_1,\ldots,\{u_1,\ldots,u_m\},\ldots,v_n)$	$\Longrightarrow$	$\{f(v_1,\ldots,u_1,\ldots,v_n),\ldots,f(v_1,\ldots,u_m,\ldots,v_n)\}$
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 $\rho_\emptyset$-calculus

- 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 $\rho_\emptyset$-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 $\rho_\emptyset$-calculus is not terminating as $[\omega](\omega)$ is a $\rho_\emptyset$-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.

3.1 The raw $\rho_\emptyset$-calculus is not confluent

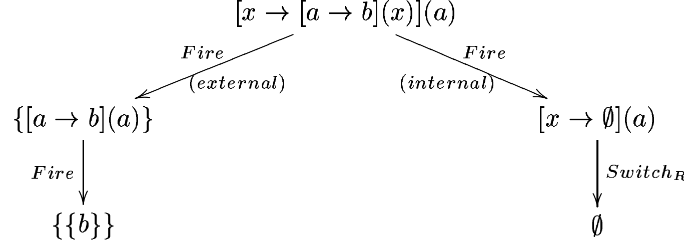
It is easy to see, and we provide typical examples just below, that the $\rho_\emptyset$-calculus is non-confluent. The main reasons for the confluence failure are due to the conflict between the use of syntactic matching and the set representation for the reductions results. This leads, on one hand, 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 and the next

one, 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 $\rho_\emptyset$-calculus that allows to restore confluence.

Let us show typical examples of confluence failure. A first such situation occurs when reducing a (sub-)term of the form $[l \to r](t)$ by matching l and t and when the matching may fail due to a free variable in t or a non-reduced sub-term of t.

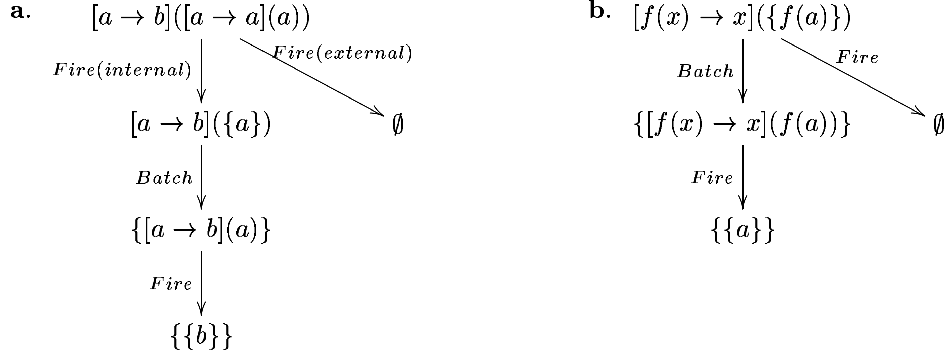
In Example 3.3 one can notice that a term can be reduced to an empty set because of a matching failure implying its bound variables. The result can be different from the empty set if the reductions of the sub-terms containing the respective variables are carried out only after the instantiation of these variables.

EXAMPLE 3.3



In Example 3.4.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 3.4.b.

EXAMPLE 3.4



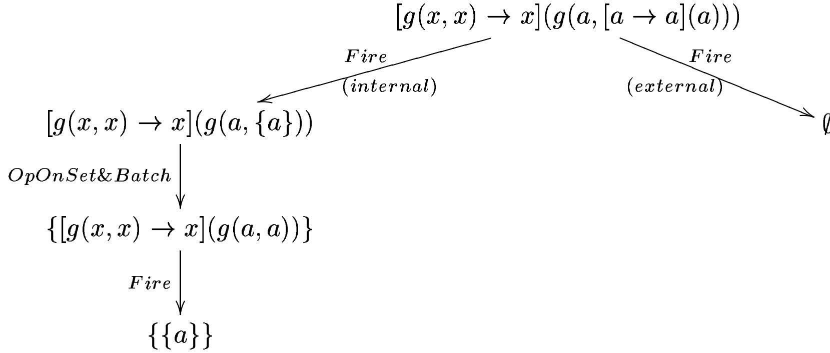
In order to avoid this kind of situation we should avoid the reduction of an application $[l \to r](t)$ if the matching between the terms l and t fails due to the matching rules *VariableClash* (Example 3.3) or *SymbolClash* (Example 3.4.a, 3.4.b) and either some variables are not instantiated or some 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 3.3 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 3.4.a and Example 3.4.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.

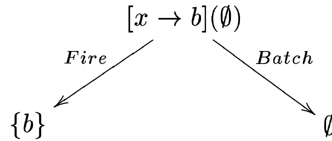
Therefore, we could consider that the evaluation rule $Fire$ is applied only when the condition on the functional positions is satisfied. Unfortunately, such a condition will not suffice for avoiding a non-appropriate matching failure due to the application of the rule $MergingClash$. As shown in Example 3.5, such a situation can be obtained if the left-hand side of the rewrite rule to be applied is not linear.

EXAMPLE 3.5



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:

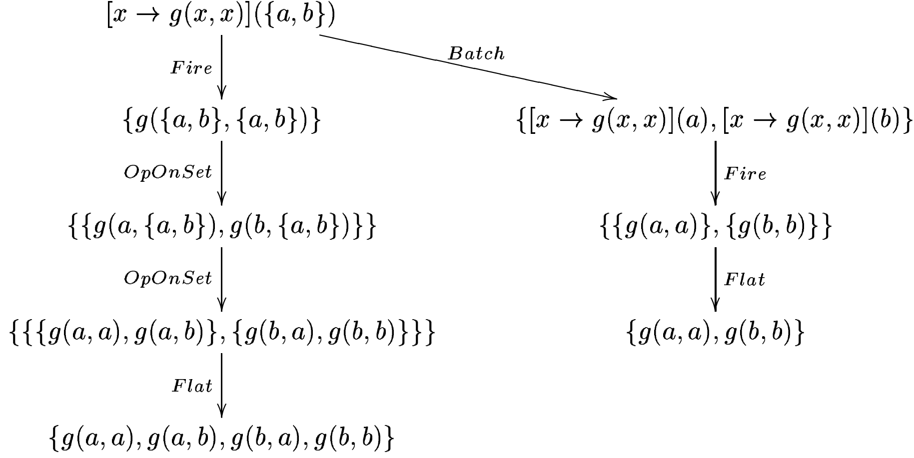
EXAMPLE 3.6



We mention that a rewrite rule is *quasi-regular* if the set of variables of the left-hand is included in the set of variables of the right-hand side. In Section 3.2 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 3.6 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 3.6) 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 3.7.

Example 3.7



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 3.6 and Example 3.7, are overcome by an eager application of the computation rules.

3.2 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 $\mathcal{NONE}$.

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 3.8
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.

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.

We propose now a strategy which emerges from the above counterexamples and which 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.

Definition 3.9
We call *ConfStratGen* 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 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 $\rho_\emptyset$-calculus such that all the sets are singletons and all the applications are of the form $[x \to u](v)$ then, all the conditions of Definition 3.9 are always satisfied. Hence, we can say that in this case the strategy *ConfStratGen* is equivalent to the strategy $\mathcal{NONE}$, *i.e.* it imposes no restriction on the reductions. We will see that the terms of the λ-calculus representation in the ρ-calculus satisfy the previous conditions and thus, the strategy *ConfStratGen* imposes no restrictions on the reductions of this instance of the ρ-calculus.

The conditions imposed in Definition 3.9 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.

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 3.10
A ρ-term l *weakly subsumes* a ρ-term t if

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

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 3.11
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 3.12
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 3.13
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 $\rho_\emptyset$-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 $\mathcal{NONE}$.

Proposition 3.14
When using the evaluation strategy *ConfStrat*, the $\rho_\emptyset$-calculus is confluent.

Proof. The proof is presented in full details in [Cir00]. □

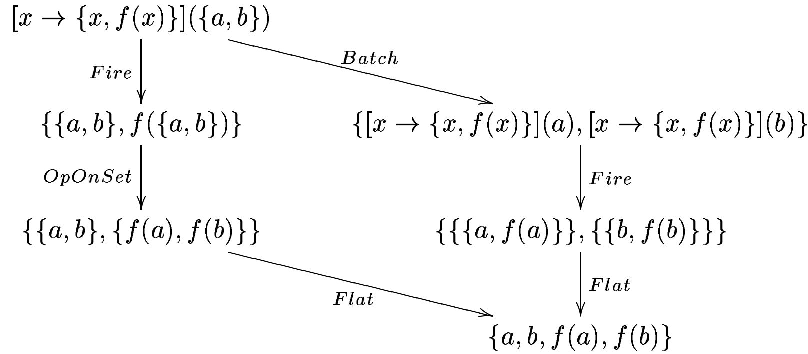
The relatively restrictive conditions imposed in strategy *ConfStrat* can be relaxed at the price of the simplicity of the strategy. The conditions that we want to weaken concern on one hand, the number of elements of the sets and on the other hand, the form of the rewrite rules.

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 3.7.

Definition 3.15
The rewrite rule $l \to r$ is *strictly 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 strictly right-linear.

The application of a rewrite rule which is not strictly right-linear to a set with more than one element can lead to non-convergent reductions, as shown in Example 3.7, but this is not the case if the applied rewrite rule is strictly right-linear:

Example 3.16



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 3.6 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 3.17
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 $t = \{u_1, \ldots, u_n\}$ then $PV(t) = \bigcap_{i=1\ldots n} PV(u_i)$, $(PV(\emptyset) = \mathcal{X})$,
3. if $t = f(u_1, \ldots, u_n)$ then $PV(t) = \bigcup_{i=1\ldots n} PV(u_i)$, $(PV(c) = \emptyset \; if \; c \in \mathcal{T}(\mathcal{F}))$,
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 3.18
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.

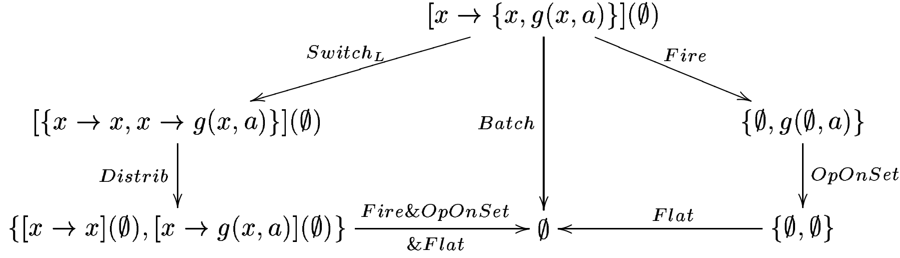
EXAMPLE 3.19
The rewrite rule $x \to g(x, y)$ is quasi-regular while the rewrite rule $x \to \{x, y\}$ is non-quasi-regular.

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 $\rho_\emptyset$-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 3.18 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 3.6.

EXAMPLE 3.20
A quasi-regular rule applied to $\emptyset$ gives only one result:



while a non-quasi-regular one yields two different results as shown in Example 3.6.

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 3.21
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 strictly 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 strictly right-linear we allow arguments containing sets having more than one element. Since one can clearly decide if a rule is quasi-regular or strictly right-linear, all the conditions used in the strategy *ConfStratLin* are decidable.

PROPOSITION 3.22
When using the evaluation strategy *ConfStratLin*, the $\rho_\emptyset$-calculus is confluent.

PROOF. A proof is given in [Cir00].
It is worth mentioning that the confluence proof of the relation induced by the evaluation rules of the $\rho_\emptyset$-calculus has a structure similar to the one followed in [CHL96] for proving the confluence of $\lambda_{\Uparrow}$.

□

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 Proposition 3.14 and Proposition 3.22 can be extended to a ρ_E-calculus modulo a specific decidable and finitary equational matching theory E.

4 Encoding λ-calculus and term rewriting in the $\rho_\emptyset$-calculus

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

4.1 Encoding the λ-calculus

We briefly present some of the notions used in the λ-calculus, 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 4.1
The β-reduction is defined by the rule:

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

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:

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

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

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

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

DEFINITION 4.2
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 $\rho_\emptyset$-calculus,
- the evaluation strategy $\mathcal{NONE}$ 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.

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

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

FIG. 7. The evaluation rules of the ρ_λ-calculus

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:

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

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:

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

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 4.3
Given two λ-terms t and t', if $t \longrightarrow_\beta t'$ then $\varphi(t) \xrightarrow{*}_{\rho_\lambda} \{\varphi(t')\}$.
Given two ρ_λ-terms u and u', if $u \longrightarrow_{\rho_\lambda} u'$ then $\delta(u) \xrightarrow{*}_\beta \delta(u')$.

PROOF. We use a structural induction on the term t:

- 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 $\varphi(u) \stackrel{*}{\longrightarrow}_{\rho_\lambda} \{\varphi(u')\}$, and thus

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

- 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

$$\varphi(t) = [\varphi(u)](\varphi(v)) \stackrel{*}{\longrightarrow}_{\rho_\lambda} [\{\varphi(u')\}](\varphi(v)) \longrightarrow_{Distrib_\lambda} \{[\varphi(u')](\varphi(v))\} = \{\varphi(t')\}.$$

  The second case is similar,

$$\varphi(t) = [\varphi(u)](\varphi(v)) \stackrel{*}{\longrightarrow}_{\rho_\lambda} [\{\varphi(u)\}](\varphi(v')) \longrightarrow_{Distrib_\lambda} \{[\varphi(u)](\varphi(v'))\} = \{\varphi(t')\}.$$

  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 4.4
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 we check that to the equality $SKK = I$ from the λ-calculus it corresponds a ρ_λ-reduction $[[S](K)](K) \stackrel{*}{\longrightarrow}_{\rho_\lambda} \{I\}$.

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

$$\{[y \to (z \to [\{y \to z\}]([y](z)))](x \to (y \to x))\} \longrightarrow_{\rho_\lambda}$$
$$\{\{[y \to (z \to [y \to z]([y](z)))](x \to (y \to x))\}\} \longrightarrow_{\rho_\lambda}$$
$$\{\{[y \to (z \to \{z\})](x \to (y \to x))\}\} \longrightarrow_{\rho_\lambda}$$
$$\{\{\{[y \to (z \to z)](x \to (y \to x))\}\}\} \longrightarrow_{\rho_\lambda}$$
$$\{\{\{\{z \to z\}\}\}\} \longrightarrow_{\rho_\lambda}$$
$$\{z \to z\} = \{I\}$$

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 is obtained independently of the reduction strategy 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 presented in Section 3.2 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 4.5
The ρ_λ-calculus is confluent.

Notice also that following the same principle, it is not difficult to see the λ-calculus with patterns of [PJ87] as a sub-calculus of the $\rho_\emptyset$-calculus.

4.2 Encoding rewriting

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 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 rewriting in the ρ-calculus is as simple as that of the λ-calculus. We consider a restriction of the $\rho_\emptyset$-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 $\rho_\emptyset$-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 $\rho_\emptyset$-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:

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

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. This representation is difficultly extended 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 4.6
Given a rewriting theory $\mathcal{T}_\mathcal{R}$ and two first order ground terms $t, t' \in \mathcal{T}(\mathcal{F})$ such that $t \stackrel{*}{\longrightarrow}_\mathcal{R} t'$. 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 $t \stackrel{*}{\longrightarrow}_\mathcal{R} t'$ such that we have $[u_n](\ldots[u_1](t)\ldots) \stackrel{*}{\longrightarrow}_{\rho\emptyset} \{t'\}$.

PROOF. We use induction on the length of the derivation $t \stackrel{*}{\longrightarrow}_\mathcal{R} t'$.

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

We have immediately $[id](t) \stackrel{0}{\longrightarrow}_{\rho\emptyset} \{t\}$.

Induction: $t \stackrel{n}{\longrightarrow}_\mathcal{R} t'$ (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,

$$t \quad \stackrel{n-1}{\longrightarrow}_\mathcal{R} \quad t'_{\lceil w\rceil_p} \quad \longrightarrow_{l\to r,p} \quad t'_{\lceil \theta r\rceil_p}$$

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 $[u_{n-1}](\ldots[u_1](t)\ldots) \stackrel{*}{\longrightarrow}_{\rho\emptyset} \{t'_{\lceil w\rceil_p}\}$. We consider the ρ-term $u_n = t'_{\lceil l\to r\rceil_p}$ and we obtain the reduction

$$\begin{array}{l}[u_n](\ldots[u_1](t)\ldots) \stackrel{*}{\longrightarrow}_{\rho\emptyset} [t'_{\lceil l\to r\rceil_p}](\{t'_{\lceil w\rceil_p}\}) \longrightarrow_{Batch} \{[t'_{\lceil l\to r\rceil_p}](t'_{\lceil w\rceil_p})\} \\ \stackrel{*}{\longrightarrow}_{Congruence} \{\{t'_{\lceil[l\to r](w)\rceil_p}\}\} \longrightarrow_{Fire} \{\{t'_{\lceil\{\theta' r\}\rceil_p}\}\} \stackrel{*}{\longrightarrow}_{OpOnSet} \{\{\{t'_{\lceil\theta' r\rceil_p}\}\}\} \\ \stackrel{*}{\longrightarrow}_{Flat} \{t'_{\lceil\theta' r\rceil_p}\}\end{array}$$

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

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

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

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

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:

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

REMARK 4.7
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 $[v;u](t)$.

5 Recursion and term traversal operators

In Section 4.2 we show that for any reduction in a rewrite theory there exists a corresponding reduction in the ρ-calculus: if the term u reduces to the term v in a rewrite theory $\mathcal{R}$ we can build a ρ-term $\xi_{\mathcal{R}}(u)$ that reduces to the term $\{v\}$. The method used for constructing the term $\xi_{\mathcal{R}}(u)$ depends on all the reduction steps from u to v in the theory $\mathcal{R}$: $\xi_{\mathcal{R}}(u)$ is a representation in the ρ-calculus of the derivation trace. We want to go further on and to give a method for constructing a term $\xi_{\mathcal{R}}(u)$ without knowing a priori the derivation from u to v. Hence we want to answer to the following question:

Given a rewrite theory $\mathcal{R}$ does there exist a ρ-term $\xi_{\mathcal{R}}$ such that for any term u, if u reduces to the term v in the rewrite theory $\mathcal{R}$ then $[\xi_{\mathcal{R}}](u)$ ρ-reduces to a set containing the term v?

This means that we wish to describe in the ρ-calculus reduction strategies and, mainly, normalization strategies. This will allow us to get, in particular, a natural encoding of normal conditional term rewriting. Therefore, we want to answer the more specific question:

Given a rewrite theory $\mathcal{R}$ does there exist a ρ-term $\xi_{\mathcal{R}}$ such that for any term u if u normalizes to the term v in the rewrite theory $\mathcal{R}$ then $[\xi_{\mathcal{R}}](u)$ ρ-reduces to a set containing the term v?

The definition of normalization strategies is in general done at the *meta-level* and an originality of our approach is to show that the ρ-calculus is powerful enough to allow us to represent such derivations at the *object level*. We will show in Section 4.1 that the $\rho_{\emptyset}$-calculus contains the λ-calculus and thus, any computable function as the normalization one is expressible in the formalism. What we bring here, because of the matching power and of the use of non-determinism, is an increased ease in the expression of such functions together with their expression in a uniform formalism combining standard rewrite techniques and higher-order behaviors.

When computing the normal form of a term u w.r.t. a rewrite system $\mathcal{R}$, the rewrite rules are applied *repeatedly* at *any position* of a term u until no rule from $\mathcal{R}$ is *applicable*. Hence, the ingredients needed for defining such a strategy are:

- an iteration operator that applies *repeatedly* a set of rewrite rules,
- a term traversal operator that applies a rewrite rule at *any position* of a term,
- an operator testing if a set of rewrite rules is *applicable* to a term.

In what follows we describe how the operators with the above functionalities can be defined in the ρ-calculus. We start with some auxiliary operators and afterwards,

we introduce the ρ-operators that correspond to the functionalities listed above.

5.1 Some auxiliary operators

First, we define three auxiliary operators that will be used in the next sections. These operators are just aliases used to define more complex ρ-terms and are used for giving more compact and clear definitions for the recursion operators.

The first of these two operators is the *identity* (denoted id) that applied to any ρ-term t evaluates to the singleton containing this term, that is $[id](t) \longrightarrow_\rho \{t\}$. The ρ-term id is nothing else but the rewrite rule $x \to x$:

$$id \triangleq x \to x.$$

In a similar way we can define the strategy $fail$ which always fails, (*i.e.* applied to any term, leads to $\emptyset$):

$$fail \triangleq x \to \emptyset.$$

The third one is the binary operator ";" that represents the sequential application of two ρ-terms. A ρ-term of the form $[u; v](t)$ represents the application of the term v to the result of the application of u to t. Therefore, we defined the operator ";" by:

$$u; v \triangleq x \to [v]([u](x)).$$

In the following sections we generally employ the abbreviations of these operators and not their expanded form but we sometimes show the corresponding reductions.

5.2 The $first$ operator

We introduce now a new operator whose role is to select between its arguments the first one that applied to a given ρ-term does not evaluate to $\emptyset$. The evaluation rules describing the $first$ operator and the auxiliary operator $\langle _, \ldots, _ \rangle$ are presented in Figure 8. We do not know currently how to express these operators in the basic ρ-calculus and we conjecture that this is not possible.

First	$[first(s_1, \ldots, s_n)](t)$	$\Longrightarrow$	$\langle [s_1](t), \ldots, [s_n](t) \rangle$
FirstFail	$\langle \emptyset, t_1, \ldots, t_n \rangle$	$\Longrightarrow$	$\langle t_1, \ldots, t_n \rangle$ if $n > 0$
FirstSuccess	$\langle t, t_1, \ldots, t_n \rangle$	$\Longrightarrow$	$\{t\}$ if t contains no redexes, no free variables and is not $\emptyset$
FirstSingle	$\langle \rangle$	$\Longrightarrow$	$\emptyset$

FIG. 8. The $first$ operator

The application of a ρ-term $first(s_1, \ldots, s_n)$ to a term t returns the result of the first "successful" application of one of its arguments to the term t. Hence, if

$[s_i](t)$ evaluates to $\emptyset$ for $i = 1, \ldots, k-1$, and $[s_k](t)$ does not evaluate to $\emptyset$, then $[first(s_1, \ldots, s_n)](t)$ evaluates to the same term as the term $[s_k](t)$.

If the evaluation of the terms $[s_i](t)$, $i = 1, \ldots, k-1$, leads to $\emptyset$ and the evaluation of $[s_k](t)$ does not terminate then the evaluation of the term $[first(s_1, \ldots, s_n)](t)$ does not terminate.

DEFINITION 5.1
The set of ρ^{1st}-terms extends the set $\varrho(\mathcal{F}, \mathcal{X})$ of basic ρ-terms (Definition 2.1), with the following two rules:

- if $t_1, \ldots, t_n$ are ρ-terms then $first(t_1, \ldots, t_n)$ is a ρ-term,
- if $t_1, \ldots, t_n$ are ρ-terms then $\langle t_1, \ldots, t_n \rangle$ is a ρ-term.

This set of terms is denoted by $\varrho^{1st}(\mathcal{F}, \mathcal{X})$.

We define now the ρ_T^{1st}-calculus by considering the new operators and the corresponding evaluation rules presented in Figure 8:

DEFINITION 5.2
Given a set $\mathcal{F}$ of function symbols, a set $\mathcal{X}$ of variables, a theory T on $\varrho^{1st}(\mathcal{F}, \mathcal{X})$ terms having a decidable matching problem, we call ρ_T^{1st}-calculus a calculus defined by:

- a non-empty subset $\varrho_{_}^{1st}(\mathcal{F}, \mathcal{X})$ of the $\varrho^{1st}(\mathcal{F}, \mathcal{X})$ terms,
- the (higher-order) substitution application to terms as defined in Section 2.2,
- a theory T,
- the set of evaluation rules $\mathcal{E}_{\rho^{1st}}$: *Fire*, *Cong*, *CongFail*, *Distrib*, *Batch*, *Switch*$_L$, *Switch*$_R$, *OpOnSet*, *Flat*, *First*, *FirstFail*, *FirstSuccess*, *FirstSingle*,
- an evaluation strategy $\mathcal{S}$ that guides the application of the evaluation rules.

The following examples present the evaluation of some ρ^{1st}-terms containing the operators of the extended calculus.

EXAMPLE 5.3
The non-deterministic application of one of the rules $a \to b$, $a \to c$, $a \to d$ to the term a is represented in the ρ-calculus by the application $[\{a \to b, a \to c, a \to d\}](a)$. This last ρ-term is reduced to the term $\{b, c, d\}$ which represents a non-deterministic choice among the three terms. If we want to apply the above rules in a deterministic way and in the specified order, we use the ρ-term $[first(a \to b, a \to c, a \to d)](a)$ with, for example, the reduction:

$$\begin{array}{ll}
 & [first(a \to b, a \to c, a \to d)](a) \\
\longrightarrow_{First} & \langle [a \to b](a), [a \to c](a), [a \to d](a) \rangle \\
\longrightarrow_{Fire} & \langle \{b\}, [a \to c](a), [a \to d](a) \rangle \\
\longrightarrow_{FirstSuccess} & \{\{b\}\} \\
\longrightarrow_{Flat} & \{b\}
\end{array}$$

We can notice that even if all the rewrite rules can be applied successfully (*i.e.* no empty set) to the term a, the final result is given by the first tried rewrite rule.

EXAMPLE 5.4
We consider now the case where some of the rules given in argument to $first$ lead to an empty set result:

$$
\begin{array}{ll}
 & [first(a \to b, b \to c, a \to d)](b) \\
\longrightarrow_{First} & \langle [a \to b](b), [b \to c](b), [a \to d](b) \rangle \\
\longrightarrow_{Fire} & \langle \emptyset, [b \to c](b), [a \to d](b) \rangle \\
\longrightarrow_{FirstFail} & \langle [b \to c](b), [a \to d](b) \rangle \\
\longrightarrow_{Fire} & \langle \{c\}, [a \to d](b) \rangle \\
\longrightarrow_{FirstSuccess} & \{\{c\}\} \\
\longrightarrow_{Flat} & \{c\}
\end{array}
$$

EXAMPLE 5.5
If none of the rules given in argument to $first$ is applied successfully, the result is obviously the empty set:

$$
\begin{array}{ll}
 & [first(a \to b, a \to c, a \to d)](b) \\
\longrightarrow_{First} & \langle [a \to b](b), [a \to c](b), [a \to d](b) \rangle \\
\stackrel{*}{\longrightarrow}_{Fire} & \langle \emptyset, \emptyset, \emptyset \rangle \\
\stackrel{*}{\longrightarrow}_{FirstFail} & \langle \rangle \\
\longrightarrow_{FirstSingle} & \emptyset
\end{array}
$$

The operator $first$ does not test explicitly the applicability of a term (rule) to another term but allows us to recover from a failure and continue the evaluation. For example, we can define a term

$$try(s) \triangleq first(s, id)$$

that applied to the term t evaluates to the result of $[s](t)$, if $[s](t)$ does not evaluate to $\emptyset$ and to $\{t\}$, if $[s](t)$ evaluates to $\emptyset$.

5.3 Term traversal operators

Let us now define operators that apply a ρ-term at some position of another ρ-term. The first step is the definition of two operators that push the application of a ρ-term one level deeper on another ρ-term. This is already possible in the ρ-calculus due to the rule *Cong* but we want to define a generic operator that applies a ρ-term r to the sub-terms u_i, $i = 1 \dots n$, of a term of the form $F(u_1, \dots, u_n)$ independently on the head symbol F.

To this end, we define two term traversal operators, $\Phi(r)$ and $\Psi(r)$, whose behavior is described by the rules in Figure 9. These operators are inspired by the operators of the *System S* described in [VeAB98].

$$
\begin{array}{lll}
TraverseSeq & [\Phi(r)](f(u_1, \dots, u_n)) & \Longrightarrow \\
 & \langle \{f([r](u_1), \dots, u_n)\}, \dots, \{f(u_1, \dots, [r](u_n))\} \rangle & \\
 & & \\
TraversePar & [\Psi(r)](f(u_1, \dots, u_n)) & \Longrightarrow \quad \{f([r](u_1), \dots, [r](u_n))\}
\end{array}
$$

FIG. 9. The term traversal operators of the ρ_T-calculus

The application of the ρ-term $\Phi(r)$ to a term $t = f(u_1, \dots, u_n)$ results in the successful application of the term r to one of the terms u_i. More precisely, r is

applied to the first u_i, $i = 1, \ldots, n$ such that $[r](u_i)$ does not evaluate to the empty set. If there *exists no* such u_i and in particular, if t is a function with no arguments (t is a constant), then the term $[\Phi(r)](t)$ reduces to the empty set.

When the ρ-term $\Psi(r)$ is applied to a term $t = f(u_1, \ldots, u_n)$ the term r is applied to all the arguments u_i, $i = 1, \ldots, n$ if *for all* i, $[r](u_i)$ does not evaluate to $\emptyset$. If there exists an u_i such that $[r](u_i)$ reduces to $\emptyset$, then the result is the empty set. If we apply $\Psi(r)$ to a constant c, since there are no sub-terms the term $[\Psi(r)](c)$ reduces to $\{c\}$.

If we consider a ρ-calculus with a finite signature $\mathcal{F}$ and if we denote by $\mathcal{F}_0 = \{c_1, \ldots, c_n\}$ the set of constant function symbols and by $\mathcal{F}_+ = \{f_1, \ldots, f_m\}$ the set of function symbols with arity at least one, the two term traversal operators can be expressed in the ρ-calculus by some appropriate ρ-terms.

If the following two definitions are considered

$$\begin{aligned}\Phi'(r) \triangleq\ & first(f_1(r, id, \ldots, id), \ldots, f_1(id, \ldots, id, r), \ldots, \\ & f_m(r, id, \ldots, id), \ldots, f_m(id, \ldots, id, r))\end{aligned}$$

$$\Psi(r) \triangleq \{c_1, \ldots, c_n, f_1(r, \ldots, r), \ldots, f_m(r, \ldots, r)\}$$

with $c_i \in \mathcal{F}_0$, $i = 1, \ldots, n$, and $f_j \in \mathcal{F}_+$, $j = 1, \ldots, m$, we obtain the following two reductions,

$$\begin{array}{ll} & [\Phi'(r)](f_k(u_1, \ldots, u_p)) \\ \triangleq & [first(f_1(r, id, \ldots, id), \ldots, f_m(id, \ldots, id, r))](f_k(u_1, \ldots, u_p)) \\ \longrightarrow_{First} & \langle [f_1(r, id, \ldots, id)](f_k(u_1, \ldots, u_p)), \ldots, [f_m(id, \ldots, id, r)](f_k(u_1, \ldots, u_p))\rangle \\ \stackrel{*}{\longrightarrow}_{Cong} & \langle \emptyset, \ldots, \emptyset, \{f_k([r](u_1), \ldots, u_p)\}, \ldots, \{f_k(u_1, \ldots, [r](u_p))\}, \emptyset, \ldots, \emptyset\rangle \\ \stackrel{*}{\longrightarrow}_{FirstFail} & \langle \{f_k([r](u_1), \ldots, u_p)\}, \ldots, \{f_k(u_1, \ldots, [r](u_p))\}, \emptyset, \ldots, \emptyset\rangle \end{array}$$

and

$$\begin{array}{ll} & [\Psi(r)](f_k(u_1, \ldots, u_p)) \\ \triangleq & [\{c_1, \ldots, c_n, f_1(r, \ldots, r), \ldots, f_m(r, \ldots, r)\}](f_k(u_1, \ldots, u_p)) \\ \longrightarrow_{Distrib} & \{[c_1](f_k(u_1, \ldots, u_p)), \ldots\ldots, [f_m(r, \ldots, r)](f_k(u_1, \ldots, u_p))\} \\ \stackrel{*}{\longrightarrow}_{Cong} & \{\emptyset, \ldots, \emptyset, \{f_k([r](u_1), \ldots, [r](u_p))\}, \emptyset, \ldots, \emptyset\} \\ \stackrel{*}{\longrightarrow}_{Flat} & \{f_k([r](u_1), \ldots, [r](u_p))\} \end{array}$$

The operator Φ' does not correspond exactly to the definition from the Figure 9 but, as we have just seen above, a similar result is obtained when applying the terms $\Phi(r)$ and $\Phi'(r)$ to a term $f_k(u_1, \ldots, u_p)$.

LEMMA 5.6
The term traversal operators Φ and Ψ can be expressed in the ρ_T^{1st}-calculus.

PROOF. If we consider $t = f_k(u_1, \ldots, u_p)$ and if for any $i = 1, \ldots, p$ we have the reductions $[r](u_i) \stackrel{*}{\longrightarrow}_\rho \emptyset$ then, according to the evaluation rules describing the behavior of $\Phi(r)$, we obtain:

$$\begin{array}{ll} & [\Phi(r)](f_k(u_1, \ldots, u_p)) \\ \longrightarrow_{TraverseSeq} & \langle \{f_k([r](u_1), \ldots, u_p)\}, \ldots, \{f_k(u_1, \ldots, [r](u_p))\}\rangle \\ \stackrel{*}{\longrightarrow}_{\rho} & \langle \{f_k(\emptyset, \ldots, u_p)\}, \ldots, \{f_k(u_1, \ldots, \emptyset)\}\rangle \end{array}$$

$$
\begin{array}{ll}
\xrightarrow{*}_{OpOnSet} & \langle\{\emptyset\},\dots,\{\emptyset\}\rangle \\
\xrightarrow{*}_{Flat} & \langle\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_{FirstFail} & \langle\rangle \\
\longrightarrow_{FirstSingle} & \emptyset
\end{array}
$$

Otherwise, if it exists an l such that $[r](u_i) \xrightarrow{*}_\rho \emptyset$, $i = 1,\dots,l-1$ and $[r](u_l) \xrightarrow{*}_\rho v_l\downarrow$, with $v_l\downarrow$ a ground term containing no redex, the following reduction is obtained:

$$
\begin{array}{ll}
 & [\Phi(r)](f_k(u_1,\dots,u_p)) \\
\longrightarrow_{TraverseSeq} & \langle\{f_k([r](u_1),\dots,u_p)\},\dots,\{f_k(u_1,\dots,[r](u_p))\}\rangle \\
\xrightarrow{*}_\rho & \langle\{f_k(\emptyset,\dots,u_p)\},\dots,\{f_k(u_1,\dots,v_l\downarrow,\dots,u_p)\},\dots,\{f_k(u_1,\dots,\emptyset)\}\rangle \\
\xrightarrow{*}_{OpOnSet} & \langle\emptyset,\dots,\emptyset,\{f_k(u_1,\dots,v_l\downarrow,\dots,u_p)\},\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_{FirstFail} & \langle\{f_k(u_1,\dots,v_l\downarrow,\dots,u_p)\},\emptyset,\dots,\emptyset\rangle
\end{array}
$$

Now, if we consider the definition of $\Phi'(r)$ and if for all $i = 1,\dots,p$ we have $[r](u_i) \xrightarrow{*}_\rho \emptyset$ then, we obtain:

$$
\begin{array}{ll}
 & [\Phi'(r)](f_k(u_1,\dots,u_p)) \\
\xrightarrow{*}_\rho & \langle\{f_k([r](u_1),\dots,u_p)\},\dots,\{f_k(u_1,\dots,[r](u_p))\},\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_\rho & \langle\{f_k(\emptyset,\dots,u_p)\},\dots,\{f_k(u_1,\dots,\emptyset)\},\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_{OpOnSet} & \langle\{\emptyset\},\dots,\{\emptyset\},\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_{Flat} & \langle\emptyset,\dots,\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_{FirstFail} & \langle\rangle \\
\longrightarrow_{FirstSingle} & \emptyset
\end{array}
$$

For the same term $[\Phi'(r)](f_k(u_1,\dots,u_p))$, if it exists an l such that $[r](u_i) \xrightarrow{*}_\rho \emptyset$, $i = 1,\dots,l-1$ and $[r](u_l) \xrightarrow{*}_\rho v_l\downarrow$, with $v_l\downarrow$ a ground term containing no redex, the following reduction is obtained:

$$
\begin{array}{ll}
 & [\Phi'(r)](f_k(u_1,\dots,u_p)) \\
\xrightarrow{*}_\rho & \langle\{f_k([r](u_1),\dots,u_p)\},\dots,\{f_k(u_1,\dots,[r](u_p))\},\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_\rho & \langle\{f_k(\emptyset,\dots,u_p)\},\dots,\{f_k(u_1,\dots,v_l\downarrow,\dots,u_p)\},\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_{OpOnSet} & \langle\{\emptyset\},\dots,\{\emptyset\},\{f_k(u_1,\dots,v_l\downarrow,\dots,u_p)\},\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_{Flat} & \langle\emptyset,\dots,\emptyset,\{f_k(u_1,\dots,v_l\downarrow,\dots,u_p)\},\emptyset,\dots,\emptyset\rangle \\
\xrightarrow{*}_{FirstFail} & \langle\{f_k(u_1,\dots,v_l\downarrow,\dots,u_p)\},\emptyset,\dots,\emptyset\rangle
\end{array}
$$

We can notice that the results of the reductions for the application of a term r to the arguments of a term $f_k(u_1,\dots,u_p)$ by using the two operators, Φ and Φ', are identical. If the terms u_i, $i = 1\dots p$, are ground terms containing no redex then, the final result of the two reductions in the case without failure is $\{f_k(u_1,\dots,v_l\downarrow,\dots,u_p)\}$.

When the operators are applied to a constant $c_k \in \mathcal{F}_0$ we obtain:

$$[\Phi'(r)](c_k) \xrightarrow{*}_\rho \langle\rangle \longrightarrow_\rho \emptyset,$$

$$[\Psi(r)](c_k) \xrightarrow{*}_\rho \{c_k\}.$$

□

5.4 Iterators

The definition of the evaluation (normalization) strategies as, for example, *top-down* or *bottom-up*, is based on the application of one term to the top position or to the deepest positions of another term.

For the moment, we have the possibility of applying a ρ-term r either to one or all the arguments u_i of a ρ-term $t = f(u_1, \ldots, u_n)$, or to the sub-terms of t at an explicitly specified depth. But the depth of a term is not known *a priori* and thus, we cannot apply a term r to the deepest positions of a term t. If we want to apply the term r to the sub-terms at the maximum depth of a term t we must define a recursive operator which reiterates the application of the $\Phi(r)$ and $\Psi(r)$ terms and thus, pushes the application deeper into terms.

We start by presenting the ρ-term used for describing recursive applications in the ρ-calculus. Starting from the fixed-point combinators of the λ-calculus, we define a ρ-term which recursively applies a given ρ-term. We use the classical fixed-point combinator of the λ-calculus ([Bar84]), $\Theta_\lambda = (A_\lambda\ A_\lambda)$ where

$$A_\lambda = \lambda xy.y(xxy)$$

and Θ_λ is called the Turing fixed-point combinator ([Tur37]).

This term corresponds in the ρ-calculus to the ρ-term $\Theta = [A](A)$ with

$$A = x \to (y \to [y]([[x](x)](y))).$$

In λ-calculus, for any λ-term G we have the reduction

$$\Theta_\lambda\ G \xrightarrow{*}_\beta G(\Theta_\lambda\ G).$$

In ρ-calculus, we have a similar reduction

$$[\Theta](G) \xrightarrow{*}_\rho \{[G]([\Theta](G))\} \qquad (\textit{Fixed Point})$$

as this can be checked as follows:

$$\begin{array}{ll}
 & [\Theta](G) \triangleq [[A](A)](G) \triangleq [[x \to (y \to [y]([[x](x)](y)))](A)](G) \\
\longrightarrow_{Fire} & [\{y \to [y]([[A](A)](y))\}](G) \\
\longrightarrow_{Distrib} & \{[y \to [y]([[A](A)](y))](G)\} \\
\longrightarrow_{Fire} & \{\{[G]([[A](A)](G))\}\} \\
\longrightarrow_{Flat} & \{[G]([[A](A)](G))\} \\
\triangleq & \{[G]([\Theta](G))\}
\end{array}$$

We have obtained the desired result but the last application of the rule *Fire* in the above reduction can be replaced by a reduction in the sub-term $[[A](A)](y)$. We can thus reduce $[[A](A)](y) \triangleq [[x' \to (y' \to [y']([[x'](x')](y')))](A)](y)$ to the term $\{[y]([[A](A)](y))\} \triangleq \{[y]([\Theta](y))\}$. We therefore obtain the following derivation:

$$\begin{array}{ll}
 & [\Theta](G) \\
\xrightarrow{*}_\rho & \{[y \to [y]([[A](A)](y))](G)\} \triangleq \{[y \to [y]([\Theta](y))](G)\} \\
\xrightarrow{*}_\rho & \{[y \to [y](\{[y]([\Theta](y))\})](G)\} \\
\xrightarrow{*}_\rho & \{[y \to [y]([y]([\Theta](y)))](G)\} \\
\xrightarrow{*}_\rho & \ldots
\end{array}$$

which does not terminate if the same redex $[\Theta](y)$ is always selected for reduction.

In an operational approach we do not want the new constructions to lead to non-terminating reductions. Since the ρ-term $[\Theta](G)$ can obviously lead to infinite reductions, a strategy should be used in order to obtain the termination and thus the desired behavior.

We should thus use a strategy which applies the evaluation rules to a sub-term of the form $[\Theta](G)$ only when no other reduction is possible. From an operational point of view, this strategy is rather difficult to implement and obviously not very efficient in a calculus where the Θ term is represented by its extended form and thus, more difficult to identify. If Θ is considered as an independent ρ-term with the behavior described by an evaluation rule corresponding to the reduction (*Fixed Point*), the strategy suggested previously could be easily implemented.

A strategy satisfying the termination condition and easier to implement could initially apply the evaluation rules at the top positions of the terms and only when no evaluation rule can be applied at the top position, reduce the sub-terms at deeper positions. It is clear that this *outermost* strategy prevents only the infinite reductions due to the operator Θ, but it cannot ensure the termination of the untyped ρ-calculus.

As we mentioned previously, the main goal of this section is the representation of normalization strategies by ρ-terms and thus, we want to describe the application of a term r to all the positions of another term t. Therefore, we must define the appropriate term G that propagates the application of a ρ-term in the sub-terms of another ρ-term.

5.4.1 Multiple applications

First, we want to define the operators *BottomUp* and *TopDown* describing the application of a term r to all the sub-terms of a term t starting with the deepest positions of t and respectively with the top position of t. We want thus to find a term which recursively applies the term r to all the sub-terms of t and afterwards at the top position of the result term and another term which initially applies the term r at the top position of the term t and then to the sub-terms of the result term. The term r must be applied to the sub-terms only if this application does not lead to a failure.

We propose first two "naive" definitions for the former operator and we comment the encountered problems. We analyze the obtained reductions and we define afterwards the operators describing the desired behavior.

The first natural possibility is to define the ρ-term

$$G_{sds}(r) \triangleq f \to (x \to [\Psi(f); r](x))$$

Let us consider the ρ-term SDS (for *SpreadDownSimple*),

$$SDS(r) \triangleq [\Theta](G_{sds}(r))$$

and its application to the term $t = f(t_1, \ldots, t_n)$. Then, the following derivation is obtained:

$$\begin{array}{ll} & [SDS(r)](t) \triangleq [[\Theta](G_{sds}(r))](t) \\ \xrightarrow{*}_{\rho} & \{[[G_{sds}(r)]([\Theta](G_{sds}(r)))](t)\} \\ \triangleq & \{[[G_{sds}(r)](SDS(r))](t)\} \end{array}$$

$$\begin{array}{ll}
\triangleq & \{[[f \to (x \to [\Psi(f); r](x))](SDS(r))](t)\} \\
\xrightarrow{*}_\rho & \{[\{x \to [\Psi(SDS(r)); r](x)\}](t)\} \\
\xrightarrow{*}_\rho & \{[\Psi(SDS(r)); r](f(t_1, \ldots, t_n))\} \\
\xrightarrow{*}_\rho & \{[r]([\Psi(SDS(r))](f(t_1, \ldots, t_n)))\} \\
\xrightarrow{*}_\rho & \{[r](f([SDS(r)](t_1), \ldots, [SDS(r)](t_n)))\}
\end{array}$$

As we can see from this derivation, the term $SDS(r)$ is recursively applied to the sub-terms of the initial term and the term r is applied at the top position of the result. If one of the applications of the term r leads to a failure, then this failure is propagated and the empty set is obtained as the result of the derivation.

When using a confluent strategy, as the ones presented in Section 3.2, the derivation presented above is possible only if the term $G_{sds}(r)$ cannot be reduced to a set with more than one element. This condition is obviously not respected if r is a set with more than one element since, for example, $G_{sds}(\{a, b\}) \xrightarrow{*}_\rho \{G_{sds}(a), G_{sds}(b)\}$. We want to prevent the evaluation of the term $G_{sds}(r)$ to a set with more than one element even when r does not satisfy this condition and therefore, we define the term

$$G_{sd}(r) \triangleq f \to (x \to \langle [\Psi(f); r](x) \rangle)$$

and respectively SD (for *SpreadDown*),

$$SD(r) \triangleq [\Theta](G_{sd}(r)).$$

If $r = \{a, b\}$ then, the term $G_{sd}(r) = G_{sd}(\{a, b\})$ is not reduced to the term $\{G_{sd}(a), G_{sd}(b)\}$ as it was the case for $G_{sds}(r)$ but

$$\begin{array}{ll}
 & G_{sd}(r) \ \triangleq \ f \to (x \to \langle [\Psi(f); \{a, b\}](x) \rangle) \\
\xrightarrow{*}_\rho & f \to (x \to \langle [\{a, b\}]([\Psi(f)](x)) \rangle) \\
\xrightarrow{*}_{Distrib} & f \to (x \to \langle \{[a]([\Psi(f)](x)), [b]([\Psi(f)](x))\} \rangle)
\end{array}$$

In this last term, the first argument of the operator $\langle\rangle$ contains the free variable x and thus, it cannot be reduced by using the evaluation rule $FirstSuccess$.

Since this last term is not a set, the propagation of the set symbols is not performed in the case of the operator G_{sd} and we can reduce the term $[\Theta](G_{sd}(r))$ to $\{[G_{sd}(r)]([\Theta](G_{sd}(r)))\}$. Consequently, we obtain the reduction:

$$\begin{array}{ll}
 & [SD(r)](t) \triangleq [[\Theta](G_{sd}(r))](t) \\
\xrightarrow{*}_\rho & \{[[G_{sd}(r)]([\Theta](G_{sd}(r)))](t)\} \\
\triangleq & \{[[G_{sd}(r)](SD(r))](t)\} \\
\triangleq & \{[[f \to (x \to \langle [\Psi(f); r](x) \rangle)](SD(r))](t)\} \\
\xrightarrow{*}_\rho & \{[\{x \to \langle [\Psi(SD(r)); r](x) \rangle\}](t)\} \\
\xrightarrow{*}_\rho & \{\langle [\Psi(SD(r)); r](f(t_1, \ldots, t_n)) \rangle\} \\
\xrightarrow{*}_\rho & \{\langle [r](f([SD(r)](t_1), \ldots, [SD(r)](t_n))) \rangle\}
\end{array}$$

EXAMPLE 5.7
If we use a strategy which initially applies the evaluation rules at the top positions of terms then, the following derivation is obtained:

$$[SD(\{a \to b, id\})](g(a, f(a)))$$

$$
\begin{array}{ll}
\xrightarrow{*}_\rho & \{\langle[\{a \to b, id\}](g([SD(\{a \to b, id\})](a), [SD(\{a \to b, id\})](f(a))))\rangle\} \\
\longrightarrow_{Distrib} & \{\langle\{[a \to b](g([SD(\{a \to b, id\})](a), [SD(\{a \to b, id\})](f(a)))), \\
& [id](g([SD(\{a \to b, id\})](a), [SD(\{a \to b, id\})](f(a))))\}\rangle\} \\
\longrightarrow_{Fire} & \{\langle\{\emptyset, [id](g([SD(\{a \to b, id\})](a), [SD(\{a \to b, id\})](f(a))))\}\rangle\} \\
\longrightarrow_{Flat} & \{\langle\{g([SD(\{a \to b, id\})](a), [SD(\{a \to b, id\})](f(a)))\}\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{g(\{\langle[\{a \to b, id\}](a)\rangle\}, [SD(\{a \to b, id\})](f(a)))\}\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{g(\{\{b, a\}, [SD(\{a \to b, id\})](f(a)))\}\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{g(\{b, a\}, \langle[\{a \to b, id\}](f([SD(\{a \to b, id\})](a)))\rangle)\}\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{g(\{b, a\}, f(\{b, a\}))\}\rangle\} \\
\xrightarrow{*}_\rho & \{g(b, f(b)), g(a, f(b)), g(b, f(a)), g(a, f(a))\}
\end{array}
$$

We can notice that the application $[SD(r)](t)$ does not guarantee that the applications of the term r to the deepest sub-terms of t are the first ones to be reduced. For example, since we try to apply the evaluation rules at the top position, in the derivation of Example 5.7 we obtain, by applying the evaluation rule $Fire$,

$$[a \to b](g([SD(\{a \to b, id\})](a), [SD(\{a \to b, id\})](f(a)))) \longrightarrow_{Fire} \emptyset$$

and not

$$[a \to b](g([SD(\{a \to b, id\})](a), [SD(\{a \to b, id\})](f(a))))$$
$$\xrightarrow{*}_\rho [a \to b](g(\{b, a\}, \{f(\{b, a\})\})) \xrightarrow{*}_\rho \emptyset$$

as in an *innermost* reduction.

The disadvantage of the non-confluence in the case of the operator SDS was eliminated by using the operator $\langle\rangle$ in the definition of the operator SD, but we have not obtained yet the desired behavior for this type of iterator. In the evaluation of the term $[SD(r)](t)$, if one of the applications of the term r to a sub-term of t is evaluated to $\emptyset$ then, this failure is propagated and the empty set is obtained as the result of the reduction.

If we want to keep unchanged the sub-terms of t on which the application of the term r evaluates to $\emptyset$, we can use the term id either in the same way as in Example 5.7, or by defining the operator G_{bu}:

$$G_{bu}(r) \triangleq f \to (x \to [first(\Psi(f), id); first(r, id)](x))$$

In the same manner as for the previous cases we obtain the operator $BottomUp$:

$$BottomUp(r) \triangleq [\Theta](G_{bu}(r))$$

corresponding to the description presented at the beginning of this section.

Lemma 5.8
The $BottomUp$ operator describing the application of a term to all the sub-terms of another term in a *bottom-up* manner can be expressed in the ρ_T^{1st}-calculus.

Proof. We analyze the reductions of the application of a term $BottomUp(r)$ to a constant and to a functional term with several arguments. A complete proof is given in [Cir00]. □

A *top-down* like reduction is immediately obtained if we take the term

$$G_{td}(r) \triangleq f \to (x \to \langle[first(r, id); first(\Psi(f), id)](x)\rangle)$$

and we define the term

$$TopDown(r) \triangleq [\Theta](G_{td}(r)).$$

LEMMA 5.9
The *TopDown* operator describing the application of a term to all the sub-terms of another term in a *top-down* manner can be expressed in the ρ_T^{1st}-calculus.

5.4.2 Singular applications

Using the term traversal operator Φ we can define similar ρ-terms that apply a specific term only at one position of a ρ-term in a *bottom-up* or *top-down* way. We will see that the operators built using the Φ operator are convenient for the construction of normalization operators.

The ρ-term used in the *bottom-up* case is

$$H_{bu}(r) \triangleq f \to (x \to [first(\Phi(f), r)](x))$$

and we define an operator that applies only once a ρ-term in a *bottom-up* way,

$$Once_{bu}(r) \triangleq [\Theta](H_{bu}(r)).$$

As for the previous operators, the term $[Once_{bu}(r)](t) \triangleq [[\Theta](H_{bu}(r))](t)$ can lead to an infinite reduction if an appropriate strategy is not employed. As for the *SpreadDown* operator it is enough to apply the evaluation rules first to the top position and only if this is not possible, to deeper positions. We can state:

LEMMA 5.10
The $Once_{bu}$ operator describing the application of a term to a sub-term of another term in a *bottom-up* manner can be expressed in the ρ_T^{1st}-calculus.

EXAMPLE 5.11
The application $[Once_{bu}(a \to b)](a)$ is reduced to $\{\langle[(a \to b)](a)\rangle\}$ and thus, to the term $\{b\}$.

The application of the rule $a \to b$ to the leftmost-innermost position of a term $g(a, f(a))$ is represented by the term $[Once_{bu}(a \to b)](g(a, f(a)))$ and the corresponding evaluation is presented below:

$$\begin{array}{ll} & [Once_{bu}(a \to b)](g(a, f(a))) \\ \xrightarrow{*}_\rho & \{\langle\langle g([Once_{bu}(a \to b)](a), f(a)), g(a, [Once_{bu}(a \to b)](f(a)))\rangle, [a \to b](g(a, f(a)))\rangle\} \\ \xrightarrow{*}_\rho & \{\langle\langle g(\{b\}, f(a)), g(a, [Once_{bu}(a \to b)](f(a)))\rangle, [a \to b](g(a, f(a)))\rangle\} \\ \xrightarrow{*}_\rho & \{\langle\{g(b, f(a))\}, [a \to b](g(a, f(a)))\rangle\} \\ \xrightarrow{*}_\rho & \{g(b, f(a))\} \end{array}$$

If we want to define an operator that applies a specific term only at one position of a ρ-term in a *top-down* way we should use the ρ-term

$$H_{td}(r) \triangleq f \to (x \to [first(r, \Phi(f))](x))$$

and we obtain immediately the operator $Once_{td}$,

$$Once_{td}(r) \triangleq [\Theta](H_{td}(r)).$$

In the case of an application $[Once_{td}(r)](t)$, the application of the term r is first tried at the top position of t and in the case of a failure, r is applied deeper in the term t. As previously, we can state:

LEMMA 5.12
The $Once_{td}$ operator describing the application of a term to a sub-term of another term in a *top-down* manner can be expressed in the ρ_T^{1st}-calculus.

5.5 Repetition and normalization operators

In the previous sections we have defined operators that describe the application of a term at some position of another term (*e.g.* $Once_{bu}$) and operators that allow us to recover from failing evaluations ($first$).

Now we want to define an operator that applies repeatedly a given strategy r to a ρ-term t. We call it *repeat* and its behavior can be described by the following evaluation rule:

$$Repeat \quad [repeat(r)](t) \implies [repeat(r)]([r](t))$$

We use once again the fixed-point operator presented in the previous section and we define the ρ-term

$$I(r) \triangleq f \to (x \to [r; f](x))$$

that is used for describing a *repeat* operator,

$$repeat(r) \triangleq [\Theta](I(r)).$$

This approach has two obvious drawbacks. First, the termination of the evaluation is not guaranteed even when the strategy used for the previous operators is used.

When the strategy applies the evaluation rules first to the top position of an application $[u](v)$ and only afterwards to the right sub-term v and then to the left sub-term u, we do not obtain the desired result. When using this *rightmost-outermost* strategy, the following non-terminating derivation is obtained:

$$[repeat(r)](t) \xrightarrow{*}_{\rho} \{[repeat(r)]([r](t))\} \xrightarrow{*}_{\rho} \dots$$

$$\xrightarrow{*}_{\rho} \{[repeat(r)]([r]([r](\dots [r](t) \dots)))\} \xrightarrow{*}_{\rho} \dots$$

Second, when the evaluation terminates the result is always the empty set. If at some point in the evaluation the application of the term r is reduced to the empty set, then $\emptyset$ is strictly propagated and thus the term $[repeat(r)](t)$ is reduced to the empty set.

In order to overcome these two problems, we can define an operator called $repeat*$ with a behavior defined by the evaluation rules presented in Figure 10.

Hence, we need an operator similar to the *repeat* one, that stores the last non-failing result and when no further application is possible returns this result. We modify the term $I(r)$ that becomes

$$J(r) \triangleq f \to (x \to [first(r; f,\ id)](x))$$

$$
\begin{array}{llll}
Repeat{*}' & [repeat{*}(r)](t) & \Longrightarrow & [repeat{*}(r)]([r](t)) \\
 & & & \text{if } [r](t) \text{ is not reduced to } \emptyset \\
Repeat{*}'' & [repeat{*}(r)](t) & \Longrightarrow & t \\
 & & & \text{if } [r](t) \text{ is reduced to } \emptyset
\end{array}
$$

FIG. 10. The operator *repeat∗*

and we define, as before, the term

$$repeat{*}(r) \triangleq [\Theta](J(r))$$

We should not forget that we assume here that an application $[u](v)$ is reduced by applying the evaluation rules at the top position, then to its argument v and only afterwards to the term u. Once again, we get:

LEMMA 5.13
The operator *repeat∗* describing the repeated application of a term while the result is not $\emptyset$ can be expressed in the ρ_T^{1st}-calculus.

EXAMPLE 5.14
The repeated application of the rewrite rules $a \to b$ and $b \to c$ on the term a is represented by the term $[repeat{*}(\{a \to b, b \to c\})](a)$ that evaluates as follows:

$$
\begin{array}{ll}
 & [repeat{*}(\{a \to b, b \to c\})](a) \\
\xrightarrow{*}_\rho & \{\langle[repeat{*}(\{a \to b, b \to c\})]([\{a \to b, b \to c\}](a)), [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{\langle[repeat{*}(\{a \to b, b \to c\})](\{b\}), [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{\langle[repeat{*}(\{a \to b, b \to c\})]([\{a \to b, b \to c\}](b)), [id](b)\rangle\}, [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{\langle[repeat{*}(\{a \to b, b \to c\})](\{c\}), [id](b)\rangle\}, [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{\langle\{\langle[repeat{*}(\{a \to b, b \to c\})]([\{a \to b, b \to c\}](c)), [id](c)\rangle\}, [id](b)\rangle\}, [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{\langle\{\langle[repeat{*}(\{a \to b, b \to c\})](\emptyset), \{c\}\rangle\}, [id](b)\rangle\}, [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{\langle\{\langle\emptyset, \{c\}\rangle\}, [id](b)\rangle\}, [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{\langle\{c\}, [id](b)\rangle\}, [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{\langle\{\{c\}\}, [id](a)\rangle\} \\
\xrightarrow{*}_\rho & \{c\}
\end{array}
$$

Using the above operators it is easy to define some specific normalization strategies. For example, the *innermost* strategy is defined by

$$im(r) \triangleq repeat{*}(Once_{bu}(r))$$

and an *outermost* strategy is defined by

$$om(r) \triangleq repeat{*}(Once_{td}(r)).$$

COROLLARY 5.15
The operators *im* et *om* describing the *innermost* and *outermost* normalization can be expressed in the ρ_T^{1st}-calculus.

We have now all the ingredients needed for describing the normalization of a term t in a rewrite theory $\mathcal{R}$. The term $\xi_{\mathcal{R}}(u)$ described at the beginning of this section can be defined using the $im(\mathcal{R})$ or $om(\mathcal{R})$ operators and thus, we can represent the normalization of a term u w.r.t. a rewriting theory $\mathcal{R}$ by the ρ-terms

$$\xi_{\mathcal{R}}(u) \triangleq [im(\mathcal{R})](u)$$

or

$$\xi_{\mathcal{R}}(u) \triangleq [om(\mathcal{R})](u).$$

EXAMPLE 5.16
If we denote by $\mathcal{R}$ the set of rewrite rules $\{a \to b, g(x, f(x)) \to x\}$, we represent by $[im(\mathcal{R})](g(a, f(a)))$ the leftmost-innermost normalization of the term $g(a, f(a))$ according to the set of rules $\mathcal{R}$ and the following derivation is obtained:

$$\begin{array}{ll}
& [im(\mathcal{R})](g(a,f(a))) \\
\triangleq & [repeat{*}(Once_{bu}(\mathcal{R}))](g(a,f(a))) \\
\xrightarrow{*}_{\rho} & \{\langle [repeat{*}(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](g(a,f(a)))), [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle [repeat{*}(Once_{bu}(\mathcal{R}))](\{g(b,f(a))\}), [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{[repeat{*}(Once_{bu}(\mathcal{R}))](g(b,f(a)))\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\{\langle [repeat{*}(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](g(b,f(a)))), \\
& \quad [id](g(b,f(a)))\rangle\}\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\langle [repeat{*}(Once_{bu}(\mathcal{R}))](\{g(b,f(b))\}), \\
& \quad [id](g(b,f(a)))\rangle\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\langle \{\langle \{[repeat{*}(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](g(b,f(b))))\}, \\
& \quad [id](g(b,f(b)))\rangle\}, [id](g(b,f(a)))\rangle\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\langle \{\langle \{[repeat{*}(Once_{bu}(\mathcal{R}))](\{b\})\}, \\
& \quad [id](g(b,f(b)))\rangle\}, [id](g(b,f(a)))\rangle\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\langle \{\langle \{\langle [repeat{*}(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](b)), [id](b)\rangle, \\
& \quad [id](g(b,f(b)))\rangle\}, [id](g(b,f(a)))\rangle\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\langle \{\langle \{\langle [repeat{*}(Once_{bu}(\mathcal{R}))](\emptyset), [id](b)\rangle, \\
& \quad [id](g(b,f(b)))\rangle\}, [id](g(b,f(a)))\rangle\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\langle \{\langle \{\langle \emptyset, [id](b)\rangle, \\
& \quad [id](g(b,f(b)))\rangle\}, [id](g(b,f(a)))\rangle\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\langle \{\langle \{\{\{b\}\}\}, [id](g(b,f(b)))\rangle\}, [id](g(b,f(a)))\rangle\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\langle \{\{b\}\}, [id](g(b,f(a)))\rangle\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{\{b\}\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{\langle \{b\}, [id](g(a,f(a)))\rangle\} \\
\xrightarrow{*}_{\rho} & \{b\}
\end{array}$$

Given a term u, if the rewriting theory $\mathcal{R}$ is not confluent then, the result of the reduction of the term $[im(\mathcal{R})](u)$ is a set representing all the possible results of the reduction of the term u in the rewriting theory $\mathcal{R}$.

EXAMPLE 5.17
Let us consider the set $\mathcal{R} = \{a \to b, a \to c, g(x, x) \to x\}$ of non-confluent rewrite rules. The term $[im(\mathcal{R})](g(a, a))$ representing the *innermost* normalization of the term $g(a, a)$ according to the set of rewrite rules $\mathcal{R}$ is reduced to $\{b, g(c, b), g(b, c), c\}$.

The term $[om(\mathcal{R})](g(a,a))$ representing the *outermost* normalization is reduced to $\{b,c\}$.

We have now all the ingredients necessary to describe in a concise way the normalization process induced by a rewrite theory. Of course, the standard properties of termination and confluence of the rewrite system will allow us to get uniqueness of the result. Our approach differs from this and we define this normalization even in the case where there is no unique normal form or where termination is not warranted. This is why in general we do not get termination or uniqueness of the normal form.

6 Using the ρ^{1st}-calculus

We have shown in Section 4.2 that a derivation in term rewriting can be mimicked into an appropriate ρ-term that indeed represents the trace of the reduction. It is often more interesting to *find* such a derivation.

6.1 Encoding rewriting in the ρ^{1st}-calculus

We are interested to build a ρ-term describing the reduction, in term rewriting, of term t w.r.t. a set of rewrite rules, but without knowing *a priori* the intermediate steps of the derivation of t. For this, we can use the ρ_T^{1st}-calculus and the operators defining *innermost* and *outermost* normalization strategies.

PROPOSITION 6.1
Given a rewriting theory $\mathcal{T}_\mathcal{R}$ and two first order ground terms $t, t\!\downarrow \in \mathcal{T}(\mathcal{F})$ such that t is normalized to $t\!\downarrow$ w.r.t. the set of rewrite rules $\mathcal{R}$. Then, $[im(\mathcal{R})](t)$ is ρ-reduced to a set containing the term $t\!\downarrow$.

PROOF. By induction on the number of reduction steps for the term t.

□

EXAMPLE 6.2
Let us consider a rewrite system $\mathcal{R}$ containing the rewrite rules $(x = x) \to True$ and $b \to a$. Then, the term $a = b$ reduces to $True$ in this rewrite system and a ρ-term reducing to $\{True\}$ can be built as shown in Section 4.2 or using the fixed-point operators.

In the former case the corresponding ρ-term is

$$[(x = x) \to True]([a = (b \to a)](a = b))$$

or

$$[(x = x) \to True](a = [b \to a](b)).$$

For the latter approach we build the term

$$[im(\{(x = x) \to True, b \to a\})](a = b).$$

Since in this case we can obtain empty sets and additionally, sets with more than one element are obtained when equational matching is not unitary, a reduction strategy as presented in Section 3.2 should be used in order to ensure the confluence.

6.2 Encoding conditional rewriting

As shown before, any term rewriting reduction can be described by a reduction in the ρ-calculus. In this section we give a representation in the ρ-calculus of the conditional rewriting reductions.

The main difficulty here resides in the fact that for conditional rewriting, the reduction relation is recursively applied in order to evaluate the condition when firing a conditional rule. We can use the same approach as our explicit description of non-conditional rewriting (see Section 4.2) but the ρ-terms used in order to describe the conditional rewriting reduction become very complicated in this case. Instead, a detailed description by a concise ρ-term of the normalization process of the conditions can be obtained by using the normalization operators presented in the Section 5.5.

6.2.1 Definition of conditional rewriting

Many conditional rewriting relations have been designed and mainly differ in the way the conditions are understood [DO90]. We consider here the normal conditional rewriting defined as follows.

DEFINITION 6.3
A *normal* rewrite system $\mathcal{R}$ is composed of conditional rewrite rules of the form $(l \to r \text{ if } c)$ where l, r, c are elements of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ with variables satisfying the condition $\mathcal{V}ar(r) \cup \mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$, and such that for each ground substitution σ satisfying $\mathcal{V}ar(c) \subseteq \mathcal{D}om(\sigma)$, the normal form under $\mathcal{R}$ of σc is either the boolean $True$ or $False$. Given a conditional rewrite system $\mathcal{R}$ composed of such rules, the application of the rewrite rule $(l \to r \text{ if } c)$ of $\mathcal{R}$ on a term t at occurrence m consists in:

(i) matching, using the substitution σ, the left-hand side of the rule against the term $t_{|m}$

(ii) normalizing the instantiated condition σc using $\mathcal{R}$ and, provided the resulting term is $True$,

(iii) replace $t_{|m}$ by σr in t.

This is denoted $t \longrightarrow_{[m]}^{l \to r \text{ if } c} t_{\lceil \sigma r \rceil_m}$.

6.2.2 Encoding

As we have mentioned, the main difficulty in the encoding of conditional rewriting is to make precise the evaluation process of the condition. In the case of normal rewriting, this means computing the normal form of the condition.

We denote by c_ρ the ρ-term that, when instantiated by the proper substitution (*i.e.* θc_ρ), normalizes to the term $\{u\}$ if the term c, instantiated accordingly (*i.e.* θc), is normalized into u in the rewrite theory $\mathcal{R}$. When the term c is a boolean condition and when the rewrite system is completely defined over the booleans [BR95], the term u should be one of the two constants $True$ or $False$.

If the reduction in a rewrite theory $\mathcal{R}$ is known, we can define, as in Section 4.2, the ρ-term $c_\rho \triangleq [u_n](\dots[u_1](c)\dots)$ that evaluates to $\{u\}$, *i.e.* to $\{True\}$ or $\{False\}$. If c_ρ is the ρ-term describing the reduction of the term c then, the conditional rewrite

rule $l \to r$ if c is represented by the ρ-term

$$l \to [\{True \to r, False \to \emptyset\}](c_\rho)$$

or even the simpler, but maybe less suggestive one,

$$l \to [True \to r](c_\rho).$$

In the case when c_ρ reduces to $\{False\}$, in the latter representation the matching fails and the result of the application is, as in the former one, the empty set. When c_ρ reduces to $\{True\}$, the result of the reduction is obviously the same in the two cases, *i.e.* the same as the application of $l \to r$.

By using the above representation, we can extend the Proposition 4.6 and show that any derivation in a conditional rewriting theory is representable by an appropriate ρ-term.

PROPOSITION 6.4
Given a conditional 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 $[u_n](\ldots[u_1](t)\ldots) \xrightarrow{*}_{\rho\emptyset} \{t'\}$.

In order to build the ρ-term c_ρ using only the term c and the rewrite rules of $\mathcal{R}$, we can use the normalization operators defined in Section 5. For example, we can define

$$c_\rho \triangleq [im(\mathcal{R})](c).$$

EXAMPLE 6.5
Let us assume that the set of rules describing the order on integers is denoted by $\mathcal{R}_<$. We consider the rewrite rule $(f(x) \to g(x)$ if $x \geq 1)$ that applied to the term $f(2)$ reduces to $g(2)$ since x is instantiated by 2 and the condition $(2 \geq 1)$ reduces to $True$ by using the rewrite rule $(2 \geq 1) \to True$.

If we consider that the condition is normalized according to $\mathcal{R}_<$, then the corresponding reduction in the ρ-calculus is the following:

$$\begin{array}{ll}
 & [f(x) \to [True \to g(x)]([im(\mathcal{R}_<)](x \geq 1))](f(2)) \\
\longrightarrow_{Fire} & \{[True \to g(2)]([im(\mathcal{R}_<)](2 \geq 1))\} \\
\xrightarrow{*}_{\rho} & \{[True \to g(2)](\{True\})\} \\
\longrightarrow_{Batch} & \{\{[True \to g(2)](True)\}\} \\
\longrightarrow_{Fire} & \{\{\{g(2)\}\}\} \\
\xrightarrow{*}_{Flat} & \{g(2)\}
\end{array}$$

The conditions of the rewrite rules can be normalized according to a set of conditional rewrite rules, including the current rule, and thus the definition of the ρ-rewrite rules representing this normalization is intrinsically recursive and cannot be realized only by using the operator im.

We use the fixed-point operator Θ described in Section 5.4 to represent the application of the same set of rewrite rules for the normalization of all the conditions.

Given a set of rewrite rules $\mathcal{R} = \mathcal{R}_n \cup \mathcal{R}_c$ where $\mathcal{R}_n$ and $\mathcal{R}_c$ represent the subset of non-conditional rewrite rules and respectively the subset of conditional rewrite rules of the form $(l \to r$ if $c)$. We define the term

$$R \triangleq f \to (y \to [im(\{l_i \to [True \to r_i]([f](c_i)) \mid i = 1 \ldots m\} \cup \mathcal{R}_n)](y))$$

where $\mathcal{R}_c = \{l_i \to r_i \text{ if } c_i \mid i = 1 \ldots m\}$, $\mathcal{R}_n = \{l'_i \to r'_i \mid i = 1 \ldots n\}$ and respectively

$$IM(R) \triangleq [\Theta](R).$$

Thus, for describing the normalization of the term t w.r.t. the rewrite rules of $\mathcal{R}$ we use the ρ-term $[IM(R)](t)$.

We obtain thus a result similar to Proposition 6.4 but with a method of construction for the corresponding ρ-term based only on the initial term and on the set of rewrite rules.

PROPOSITION 6.6
Given a conditional rewriting theory $\mathcal{T}_{\mathcal{R}}$ and two first order ground terms $t, t{\downarrow} \in \mathcal{T}(\mathcal{F})$ such that t is normalized to $t{\downarrow}$ w.r.t. the set of rewrite rules $\mathcal{R}$. Then, $[IM(\mathcal{R})](t)$ is ρ-reduced to a set containing the term $t{\downarrow}$.

EXAMPLE 6.7
We consider the set of rewrite rules $\mathcal{R}$ containing the rewrite rule $(x = x) \to True$ and the conditional rewrite rules $(f(x) \to g(x) \text{ if } h(x) = b)$ and $(h(x) \to b \text{ if } x = a)$. The term $f(a)$ reduces to $g(a)$ using the rewrite rules of $\mathcal{R}$ and we show below the corresponding reduction in ρ-calculus.

Using the method presented above we obtain the ρ-term:

$$\begin{array}{l} R \triangleq f \to (y \to [im(\{f(x) \to [True \to g(x)]([f](h(x) = b)), \\ \qquad\qquad\qquad h(x) \to [True \to b]([f](x = a)), \\ \qquad\qquad\qquad (x = x) \to True \\ \qquad\qquad \})](y)) \end{array}$$

We show the main steps in the reduction of the term $[IM(R)](f(a))$. We obtain immediately the reduction

$$[IM(R)](f(a)) \triangleq [[\Theta](R)](f(a)) \quad \stackrel{*}{\longrightarrow}_\rho \quad [[R]([\Theta](R))](f(a)) \triangleq [[R](IM(R))](f(a))$$

and the final result is the same as the one obtained for the term

$$\begin{array}{l} [im(\{f(x) \to [True \to g(x)]([IM(R)](h(x) = b)), \\ \quad h(x) \to [True \to b]([IM(R)](x = a)), \\ \quad (x = x) \to True \\ \})](f(a))) \end{array}$$

and thus for

$$[f(x) \to [True \to g(x)]([IM(R)](h(x) = b))](f(a))$$
$$\stackrel{*}{\longrightarrow}_\rho \quad \{[True \to g(a)]([IM(R)](h(a) = b))\}$$

For the term $[IM(R)](h(a) = b)$ we proceed as previously and thus, we have to reduce the term

$$\begin{array}{l} [im(\{f(x) \to [True \to g(x)]([IM(R)](h(x) = b)), \\ \quad h(x) \to [True \to b]([IM(R)](x = a)), \\ \quad (x = x) \to True\} \\ )](h(a) = b) \end{array}$$

with the intermediate reduction

$$[h(x) \to [True \to b]([IM(R)](x = a))](h(a)) \quad \stackrel{*}{\longrightarrow}_\rho \quad \{[True \to b]([IM(R)](a = a))\}$$

Since we easily obtain $[IM(R)](a = a) \xrightarrow{*}_\rho \{True\}$ then, the previous term is reduced to $\{[True \to b](\{True\})\} \xrightarrow{*}_\rho \{b\}$ and we have

$$[IM(R)](h(a) = b) \quad \xrightarrow{*}_\rho \quad [im(\dots)](\{b\} = b) \quad \xrightarrow{*}_\rho \quad \{True\}$$

We come back to the reduction of the initial term and we get

$$\{[True \to g(a)]([IM(R)](h(a) = b))\} \quad \xrightarrow{*}_\rho \quad \{[True \to g(a)](\{True\})\} \quad \xrightarrow{*}_\rho \quad \{g(a)\}$$

We have thus obtained the same result as in conditional term rewriting.

Starting from the results presented in this section we will give in the next section a representation of the more elaborated rewrite rules used in ELAN, a language based on conditional rewrite rules with local assignments.

7 The rewriting calculus as a semantics of ELAN

7.1 ELAN*'s rewrite rules*

ELAN is an environment for specifying and prototyping deduction systems in a language based on labeled conditional rewrite rules and strategies to control rule application. The ELAN system offers a compiler and an interpreter of the language. The ELAN language allows us to describe in a natural and elegant way various deduction systems [Vit94, KKV95, BKK+96]. It has been experimented on several non-trivial applications ranging from decision procedures, constraint solvers [Cas98], logic programming [KR98] and automated theorem proving [CK97] but also specification and exhaustive verification of authentication protocols [Cir99].

ELAN's rewrite rules are conditional rewrite rules with local assignments. The local assignments are let-like constructions that allow applications of strategies to some terms. The general syntax of an ELAN rule is:

$$[\ell] \quad l \Rightarrow r \qquad [\ \textbf{if}\ \ cond \ \ \mid \ \ \textbf{where}\ \ y := (S)u\ ]^* \quad end$$

We should notice that the square brackets ([ ]) in ELAN are used to indicate the label of the rule and should be distinguished from the square brackets of the ρ-calculus that represent the application of a rewrite rule (ρ-term).

A partial semantics could be given to an ELAN program using rewriting logic [Mes92, BKKM99], but more conveniently ELAN's rules can be expressed using the ρ-calculus and thus an ELAN program is just a set of ρ-terms.

EXAMPLE 7.1

An example of an ELAN rule describing a possible naive way to search the minimal element of a list by sorting the list and taking the first element is the following:

```
[min-rule]  min(l)    =>   m
                 if l !=  nil
                 where sl := (sort) l
                 where m  := () head(sl)  end
```

The strategy sort can be any sorting strategy. The operator head is supposed to be described by a confluent and terminating set of unlabeled rewrite rules.

The evaluation strategy used for evaluating the conditions is a leftmost innermost standard rewriting strategy.

The non-determinism is handled mainly by two basic strategy operators: `dont care choose` (denoted $\mathtt{dc}(s_1, \ldots, s_n)$) that returns the results of at most one non-deterministicly chosen unfailing strategy from its arguments and `dont know choose` (denoted $\mathtt{dk}(s_1, \ldots, s_n)$) that returns all the possible results. A variant of the `dont care choose` operator is the `first choose` operator (denoted $\mathtt{first}(s_1, \ldots, s_n)$) that returns the results of the first unfailing strategy from its arguments.

Several strategy operators implemented in ELAN allow us a simple and concise description of user defined strategies. For example, the concatenation operator denoted ";" builds the sequential composition of two strategies s_1 and s_2. The strategy $s_1; s_2$ fails if s_1 fails, otherwise it returns all results (maybe none) of s_2 applied to the results of s_1. Using the operator `repeat*` we can describe the repeated application of a given strategy. Thus, $\mathtt{repeat*}(s)$ iterates the strategy s until it fails and then returns the last obtained result.

Any rule in ELAN is considered as a basic strategy and several other strategy operators are available for describing the computations. Here is a simple example illustrating the way the `first` and `dk` strategies work.

Example 7.2
If the strategy `dk(x=>x+1,x=>x+2)` is applied to the term a, ELAN provides two results: $a+1$ and $a+2$. When the strategy `first(x=>x+1,x=>x+2)` is applied to the same term only the $a+1$ result is obtained. The strategy `first(b=>b+1,a=>a+2)` applied to the term a yields the result $a+2$.

Using non-deterministic strategies, we can explore exhaustively the search space of a given problem and find paths described by some specific properties.

For example, for proving the correctness of the Needham-Schroeder authentication protocol [NS78] we look for possible attacks among all the behaviors during a session. In Example 7.3 we present just one of the rules of the protocol and we give the strategy looking for all the possible attacks, a more detailed description of the implementation is given in [Cir99].

Example 7.3
The Needham-Schroeder authentication protocol aims to establish a mutual authentication between an initiator and a responder that communicate via an insecure network (*i.e.* in presence of intruders).

The rules of the protocol describe the change of the global state for a session. The global state consists of the state of the sender, the state of the responder, the state of the intruder and the messages in the network.

The rule `initiate` initiates the session: the sender identified by the variable `x` and whose local state is `SLEEP` sends the message created with the function `createMessage` to the responder `y`, that is also in the state `SLEEP`. The message `messXY` is sent by adding it at the beginning of the list `Net` representing the network. The nonce `N(x,y)` (a number that identifies the session) sent in the message is stored by the initiator for further verifications. Once the message is sent, the initiator changes its local state to `WAIT` and waits for an acknowledgement.

```
[initiate]
x+SLEEP+noncex  <>  y+SLEEP+noncey  <>  Intruder  <>  Net          =>
x+WAIT+N(x,y)   <>  y+SLEEP+noncey  <>  Intruder  <>  messXY . Net
                          where messXY :=() createMessage(x,y)
end
```

Several other rewrite rules describe the other rules of the protocol and the behavior of the intruder.

The strategy looking for possible attacks applies repeatedly and non-deterministicly all the rewrite rules describing the behavior of the protocol and of the intruder and selects only those results representing an attack.

```
[]attStrat => repeat*(
                        dk( initiate, ..., intruder)
              );
              attackFound      end
```

The non-deterministic application is described with the operator `dk`. The result of the strategy `repeat*(...)` is the set of all possible behaviors in a protocol session where messages can be intercepted or faked by an intruder. The strategy `attackFound` just checks if the term received as input represents an attack (by trying to apply the rewrite rules corresponding to the negation of the desired invariants) and therefore selects from the previous set of results only those representing an attack.

7.2 The ρ-calculus representation of ELAN *rules*

The rules of the system ELAN can be expressed using the ρ-calculus. A rule with no conditions and no local assignments $l \Rightarrow r$ is represented by $l \to r$ and a conditional rule is expressed as in Section 6.2.

7.2.1 Rules with local assignments

The ELAN rewrite rules with local assignments but without conditions of the form

$$[\ell] \quad l(x) \Rightarrow \quad r(x,y)$$
$$\texttt{where } y := (S)u$$

can be represented by the ρ-term

$$l(x) \to r(x, [S_\rho](u))$$

or the ρ-term

$$l(x) \to [y \to r(x,y)]([S_\rho](u))$$

with S_ρ, the ρ-term corresponding to the strategy S in the ρ-calculus.

The first representation syntactically replaces all variables of the right-hand side of the rewrite rule defined in a local assignment with the term which instantiates the respective variable. In the second representation, each variable defined in a local assignment is bound in a ρ-rewrite rule which is applied to the corresponding term.

Example 7.4
The ELAN rule

```
[deriveSum] p_1 + p_2  =>  p_1' + p_2'
                            where p_1' := (derive)p_1
                            where p_2' := (derive)p_2       end
```

can be represented by one of the following two ρ-terms
$p_1 + p_2 \rightarrow [derive](p_1) + [derive](p_2)$,
$p_1 + p_2 \rightarrow [p'_1 \rightarrow [p'_2 \rightarrow p'_1 + p'_2]([derive](p_2))]([derive](p_1))$.

At this moment one can notice the usefulness of free variables in the rewrite rules. The latter representation of an ELAN rule with local assignments would not be possible if the variable p'_1 was not allowed to be free in the ρ-rule $p'_2 \rightarrow p'_1 + p'_2$. The free variables in the right-hand side of a ρ-rewrite-rule also enables the parameterization of rewrite rules by strategies as in $y \rightarrow [f(x) \rightarrow [y](x)](f(a))$ where the strategy to be applied on x is not known in the rule $f(x) \rightarrow [y](x)$.

Example 7.5
We consider the ELAN rule

```
[deriveSum] x  =>  y + y
                   where y := (derive)x       end
```

Let us consider that the strategy `derive` is `dk(a=>b,a=>c)`. Then, the application of the strategy `derive` to the term a gives the two results b and c. Thus, the application of the rule `deriveSum` to the term a provides non-deterministically one of the four results $b+b$, $b+c$, $c+b$, $c+c$.

The ρ-representation of this rule is

$$x \rightarrow [\{a \rightarrow b, a \rightarrow c\}](x) + [\{a \rightarrow b, a \rightarrow c\}](x)$$

that applied to a reduces as follows

$$\begin{array}{ll}
 & [x \rightarrow [\{a \rightarrow b, a \rightarrow c\}](x) + [\{a \rightarrow b, a \rightarrow c\}](x)](a) \\
\longrightarrow_{Fire} & \{[\{a \rightarrow b, a \rightarrow c\}](a) + [\{a \rightarrow b, a \rightarrow c\}](a)\} \\
\stackrel{*}{\longrightarrow}_{Distrib} & \{\{[a \rightarrow b](a), [a \rightarrow c](a)\} + \{[a \rightarrow b](a), [a \rightarrow c](a)\}\} \\
\stackrel{*}{\longrightarrow}_{Fire} & \{\{\{b\}, \{c\}\} + \{\{b\}, \{c\}\}\} \\
\longrightarrow_{Flat} & \{\{b, c\} + \{b, c\}\} \\
\longrightarrow_{OpOnSet} & \{\{b + \{b, c\}, c + \{b, c\}\}\} \\
\longrightarrow_{OpOnSet} & \{\{\{b + b, b + c\}, \{c + b, c + c\}\}\} \\
\longrightarrow_{OpOnSet} & \{\{\{b + b, b + c\}, \{c + b, c + c\}\}\} \\
\stackrel{*}{\longrightarrow}_{Flat} & \{b + b, b + c, c + b, c + c\}
\end{array}$$

This set represents exactly the four results obtained in ELAN.

If we consider more general ELAN rules containing local assignments as well as conditions on the local variables, the combination of the methods used for conditional rules and rules with local assignments should be done carefully. If we had used a representation closed to the first one from Example 7.4 we would have obtained some incorrect results as in Example 7.6.

Example 7.6
We consider the description of an automaton by a set of rewrite rules, each one describing the transition from a state to another. The potential execution of a double transition from an initial state in a final state passing by a non-final intermediate state, can be described by the following ELAN rule:

```
[double] x => next(y)
                   where y := (dk(s1 => s2,s1 => s3)) x
                   if nf(y)
end
```

The term `next(y)` represents the state obtained by carrying out a transition from `y` and this behavior can be easily represented in ELAN by a set of unlabeled rules describing the operator `nf`. We note by $\mathcal{R}_f$ the set of rewrite rules describing the final states and we suppose that `s2` is a final state but `s3` is not.

By using the first representation approach of a rule with local assignments and the coding method for conditional rules presented in Section 6.2, we obtain the ρ-term corresponding to the previous ELAN rule:

$$x \to [True \to next([\{s1 \to s2, s1 \to s3\}](x))]([im(\mathcal{R}_f)](nf([\{s1 \to s2, s1 \to s3\}](x))))$$

This term applied to $s1$ leads to the following reduction

$$\begin{array}{l}
[x \to [True \to next([\{s1 \to s2, s1 \to s3\}](x))]([im(\mathcal{R}_f)](nf([\{s1 \to s2, s1 \to s3\}](x))))](s1) \\
\longrightarrow_\rho \{[True \to next([\{s1 \to s2, s1 \to s3\}](s1))]([im(\mathcal{R}_f)](nf([\{s1 \to s2, s1 \to s3\}](s1))))\} \\
\stackrel{*}{\longrightarrow}_\rho \{[True \to next(\{s2, s3\})]([im(\mathcal{R}_f)](nf(\{s2, s3\})))\} \\
\stackrel{*}{\longrightarrow}_\rho \{[True \to \{next(s2), next(s3)\}]([im(\mathcal{R}_f)](\{nf(s2), nf(s3)\}))\} \\
\stackrel{*}{\longrightarrow}_\rho \{[True \to \{next(s2), next(s3)\}](\{False, True\})\} \\
\stackrel{*}{\longrightarrow}_\rho \{\{[True \to \{next(s2), next(s3)\}](False), [True \to \{next(s2), next(s3)\}](True)\}\} \\
\stackrel{*}{\longrightarrow}_\rho \{\emptyset, [True \to \{next(s2), next(s3)\}](True)\} \\
\stackrel{*}{\longrightarrow}_\rho \{\emptyset, \{next(s2), next(s3)\}\} \\
\stackrel{*}{\longrightarrow}_\rho \{next(s2), next(s3)\}
\end{array}$$

while in ELAN we obtain the only result `next(s3)` that would be represented by the ρ-term $\{next(s3)\}$.

The problem in the Example 7.6 is the double evaluation of the term $[\{s1 \to s2, s1 \to s3\}](s1)$ replacing the local variable y: once in the condition and once in the right-hand side of the rule. If this term is evaluated to a set with more than one element and one of its elements satisfies the condition, then this set replaces the corresponding variables in the right-hand side of the rule, while only the subset of elements satisfying the condition should be considered. Therefore, we need a mechanism that evaluates only once each of the local assignments of a rule.

We use an approach combining the second representation approach of a rule with local assignments and the ρ-representation of conditional rules. Without losing generality, we consider that an ELAN rule that has the following form:

$[label] \quad l \Longrightarrow r_{\lceil x \rceil_q}$
    `where` $x := (s)t$
    `if` $C_{\lceil x \rceil_p}$
`end`

Then, the ELAN rule presented above is expressed as the ρ-term

$$l \to [x \to [\{True \to r_{\lceil x \rceil_q}, False \to \emptyset\}]([im(\mathcal{R})](C_{\lceil x \rceil_p}))]([s](t))$$

or the simpler one

$$l \to [x \to [True \to r_{\lceil x \rceil_q}]([im(\mathcal{R})](C_{\lceil x \rceil_p}))]([s](t))$$

where $\mathcal{R}$ represents the set of rewrite rules modulo which we normalize the conditions.

In order to simplify the presentation we supposed that the rules of the set $\mathcal{R}$ are rewrite rules of the form $l \to r$ and thus, the operator im is sufficient to define normalization w.r.t. such a set. If we consider conditional unlabeled rules, then the operator IM must be employed.

The way the transformation is applied to an ELAN rewrite rule and the corresponding reduction are illustrated by taking again the Example 7.6 and considering the new representation.

EXAMPLE 7.7
The ELAN rewrite rule from Example 7.6 is represented by the ρ-term

$$x \to [y \to [True \to next(y)]([im(\mathcal{R}_f)](nf(y)))]([\{s1 \to s2, s1 \to s3\}](x))$$

that, applied to the term $s1$ leads to the following reduction

$$\begin{array}{ll}
\multicolumn{2}{l}{[x \to [y \to [True \to next(y)]([im(\mathcal{R}_f)](nf(y)))]([\{s1 \to s2, s1 \to s3\}](x))](s1)} \\
\longrightarrow_{Fire} & \{[y \to [True \to next(y)]([im(\mathcal{R}_f)](nf(y)))]([\{s1 \to s2, s1 \to s3\}](s1))\} \\
\stackrel{*}{\longrightarrow}_{\rho} & \{[y \to [True \to next(y)]([im(\mathcal{R}_f)](nf(y)))](\{s2, s3\})\} \\
\stackrel{*}{\longrightarrow}_{\rho} & \{[y \to [True \to next(y)]([im(\mathcal{R}_f)](nf(y)))](s2), \\
 & [y \to [True \to next(y)]([im(\mathcal{R}_f)](nf(y)))](s3)\} \\
\stackrel{*}{\longrightarrow}_{Fire} & \{\{[True \to next(s2)]([im(\mathcal{R}_f)](nf(s2)))\}, \\
 & \{[True \to next(s3)]([im(\mathcal{R}_f)](nf(s3)))\}\} \\
\stackrel{*}{\longrightarrow}_{\rho} & \{[True \to next(s2)](False), [True \to next(s3)](True)\} \\
\stackrel{*}{\longrightarrow}_{\rho} & \{\emptyset, \{next(s3)\}\} \\
\stackrel{*}{\longrightarrow}_{\rho} & \{next(s3)\}
\end{array}$$

that is the representation of the result obtained in ELAN.

The same result as in Example 7.6 is obtained if the evaluation rule $Fire$ is applied before the distribution of the set $\{s2, s3\}$. But the confluent strategies presented in Section 3.2 forbid such a reduction and thus, the correct result is obtained.

This latter representation not only allows a correct transformation of ELAN reductions in ρ-reductions but gives also a hint on the implementation details of such rewrite rules. On one hand the implementation should ensure the correctness of the result and on the other hand it should take into account the efficiency problems. For

instance, the representation used in Example 7.5 is correct but obviously less efficient than a representation as in Example 7.7 and this is due to the double evaluation of the same application.

The ELAN evaluation mechanism is more complex than presented above. In ELAN we distinguish between labeled rewrite rules and unlabeled rewrite rules. The unlabeled rewrite rules are used to normalize the result of all the applications of a labeled rewrite rule to a term. When evaluating a local assignment `where v:=(S) t` of an ELAN rewrite rule, the term `t` is first normalized according to the specified set of unlabeled rewrite rules and then the strategy `S` is applied to its normal form. Moreover, each time a labeled rewrite rule is applied to a term, the ELAN evaluation mechanism normalizes the result of its application with respect to the set of unlabeled rewrite rules.

Hence, the ELAN rewrite rule from Example 7.6 should be represented in the ρ-calculus by the term

$$x \to [im(\mathcal{R}_f)]([y \to [True \to next(y)] \\ ([im(\mathcal{R}_f)](nf(y)))]([\{s1 \to s2, s1 \to s3\}]([im(\mathcal{R}_f)](x))))$$

where $\mathcal{R}_f$ represents the set of (unlabeled) rewrite rules modulo which we normalize the local assignments.

7.2.2 General strategies in the local assignments

Until now we have considered in the local assignments of a rule only strategies that do not use the respective rewrite rule. The representation of an ELAN rule with local calls to strategies defined by using this rule must be parameterized by the definition of the respective strategies. For example, a rule with local assignments of the form

$$[label] \quad l \Longrightarrow r \\ \texttt{where } x := (s)t$$

is represented by the ρ-term

$$label(f) \triangleq l \to [x \to r]([[f](s)](t))$$

where the free variable f will be instantiated by the set of strategies of the program containing the rule labeled by *label*.

7.2.3 ELAN strategies and programs

The elementary ELAN strategies has, in most of the cases, a direct representation in the ρ-calculus. The identity (`id`) and the failure (`fail`) as well as the concatenation (`;`) are directly represented in the ρ-calculus by the ρ-operators id, $fail$ and ";" respectively, defined in Section 5.1. The strategy $\texttt{dk}(S_1, \ldots, S_n)$ is represented in the ρ-calculus by the set $\{S_1, \ldots, S_n\}$ and the strategy $\texttt{first}(S_1, \ldots, S_n)$ by the ρ-term $first(S_1, \ldots, S_n)$ defined in Section 5.2. The iteration strategy operator `repeat*` is easily represented by using the ρ-operator $repeat*$.

Strategies can be used in the evaluation of the local assignments and these strategies are expressed using rewrite rules. Therefore, the ELAN strategies can be represented by ρ-terms in the same way as the ELAN rewrite rules.

Example 7.8
The ELAN strategy `attStrat` used in Example 7.3 is immediately represented by the ρ-term

$$attStrat_\rho \to repeat*(\{initiate_\rho, \ldots, intruder_\rho\}); attackFound_\rho$$

where we suppose that $initiate_\rho$, $intruder_\rho$, $attackFound_\rho$ are the representations in ρ-calculus of the corresponding ELAN strategies.

For the representation of the user-defined strategies in an ELAN program we use an approach based on the fixed-point operator and similar to that used in the case of conditional rules in Section 6.2. If we consider an ELAN program containing the strategies $S_1, \ldots, S_n$ and a set of labeled rules, then the ρ-term representing the program is

$$P \triangleq [\Theta](S)$$

where

$$S \triangleq f \to (y \to [\{S_i \to Body_i \mid i = 1 \ldots n\}](y))$$

and $Body_i$ represent the right-hand sides of the strategies with each strategy S_i replaced by $[f](S_i)$, each rule label replaced by the ρ-representation of the rule and each ELAN strategy operator replaced by its correspondent in the ρ-calculus.

To sum-up, we present the transformation of an ELAN program in a ρ-term.

Definition 7.9
We consider an ELAN without importations.

1. The signature of the corresponding ρ-calculus is obtained from the operator declarations of the ELAN program.
2. Starting from unlabeled rules of the form

   $$[] \quad l_i(\overline{x}) \Longrightarrow r_i(\overline{x}, \overline{y}) \quad \texttt{where } (sort)\ u_i(\overline{y}) := ()t_i(\overline{x}) \quad \texttt{if } c_i(\overline{x}, \overline{y}) \quad \texttt{end}$$

   we build the term

   $$R_{nn} \triangleq f \to (z \to [im(\{l_i(\overline{x}) \to [u_i(\overline{y}) \to [True \to r_i(\overline{x}, \overline{y})]([f](c_i(\overline{x}, \overline{y})))](t_i(\overline{x})) \mid i = 1 \ldots n\})](z))$$

   The *innermost* normalization w.r.t. the set of unlabeled rules is represented by the term

   $$IM_{nn} \triangleq [\Theta](R_{nn})$$

   The encoding is extended in an incremental way to rules containing several conditions and local assignments. The encoding can be simplified if the program does not contain unlabeled conditional rules; in this case the term IM_{nn} becomes

   $$IM_{nn} \triangleq im(\{l_i(\overline{x}) \to [u_i(\overline{y}) \to r_i(\overline{x}, \overline{y})](t_i(\overline{x})) \mid i = 1 \ldots n\})$$

   where the rules with local assignments can be simplified to elementary rules.

3. For each labeled rule of the form

$$
\begin{array}{lll}
[label] & l(\overline{x}) \Longrightarrow r(\overline{x},\overline{y}) & \\
& & \texttt{where } (sort)\ u(\overline{y}) := (s)t(\overline{x}) \\
& & \texttt{if } c(\overline{x},\overline{y}) \\
\texttt{end} & &
\end{array}
$$

we build the term

$$
label(f) \triangleq f \to (l(\overline{x}) \to [IM_{nn}]([u(\overline{y}) \to [True \to r(\overline{x},\overline{y})]([IM_{nn}](c(\overline{x},\overline{y})))]([[f](s)]([IM_{nn}](t(\overline{x}))))))
$$

4. For each strategy of the form

$$
\begin{array}{ll}
[] & S \Longrightarrow Body \\
\texttt{end} &
\end{array}
$$

we build the term

$$
S \to BodyRho(f)
$$

where $BodyRho$ represents the right-hand side $Body$ of the strategy with each strategy symbol S_i replaced by $[f](S_i)$, each rule label $label$ replaced by the ρ-representation $label(f)$ of the rule and each ELAN strategy operator replaced by its correspondent in the ρ-calculus.

The ELAN program defining the strategies $S_1, \ldots, S_n$ is represented by the ρ-term

$$
P \triangleq [\Theta](S)
$$

where

$$
S \triangleq f \to (z \to [\{S_i \to BodyRho_i(f) \mid i = 1 \ldots n\}](z))
$$

and $BodyRho_i(f)$ represents the encoding of the strategy S_i.

The application of a strategy $\mathcal{S}$ of an ELAN program $\mathcal{P}$ to a term t is represented by the ρ-term $[[P](s)](t)$ where P is the ρ-term representing the program $\mathcal{P}$ and s is the name of the strategy $\mathcal{S}$. If the execution of the program $\mathcal{P}$ for evaluating the term t according to the strategy $\mathcal{S}$ leads to the results $u_1, \ldots, u_n$, then the ρ-term $[[P](s)](t)$ is reduced to the set term $\{u_1, \ldots, u_n\}$.

In Example 7.10 we present an ELAN module and the ρ-interpretations of all the rules and strategies and thus, of the ELAN program.

EXAMPLE 7.10
The module `automaton` describes an automaton with the states `s1,s2,s3,s4,s5` and with the non-deterministic transitions described by a set of rules containing the rules labeled with `r12,r13,r25,r32,r34,r41`. The operator `next` defines the next state in a deterministic manner and its behavior is described by a set of unlabeled rules. The states can be "final" (`final`) or "closed" (`closed`). The double transitions with an intermediate non-final and non-closed state are described by the rules `double_f` and respectively `double_c`.

```
module  automaton
import global bool;end
sort state ;end
operators global
    s1,s2,s3,s4,s5 : state;
    next(@) : (state)  state;
    final(@) : (state)  bool;
    closed(@) : (state)  bool;
end
stratop  global
    follow   : <state -> state>   bs;
    gen_double   : <state -> state>   bs;
    cond_double   : <state -> state>   bs;
end
rules for bool
global
 [] final(s_1) => false   end         [] closed(s_1) => false   end
 [] final(s_2) => true    end         [] closed(s_2) => false   end
 [] final(s_3) => false   end         [] closed(s_3) => true    end
 [] final(s_4) => false   end         [] closed(s_4) => true    end
 [] final(s_5) => true    end         [] closed(s_5) => true    end
end
rules for state
    x,y : state;
global
  [r12] s1 => s2   end                  [] next(s1) => s3   end
  [r13] s1 => s3   end                  [] next(s2) => s5   end
  [r25] s2 => s5   end                  [] next(s3) => s2   end
  [r32] s3 => s2   end                  [] next(s4) => s1   end
  [r34] s3 => s4   end                  [] next(s5) => s5   end
  [r41] s4 => s1   end

  [double_f]  x => next(y)
                     where y := (follow) x
                     if not final(y)                end
  [double_c]  x => next(y)
                     where y := (follow) x
                     if not closed(y)               end
end

strategies for state
implicit
    []follow        =>   dk(r12,r13,r25,r32,r34,r41)            end
    []gen_double    =>   follow;follow                          end
    []cond_double   =>   dk(double_f,double_c)                  end
end
end
```

We denote by B the set of unlabeled rules defined in the imported modules `bool` and describing operations on booleans.

The set of unlabeled rules from the module `automaton` are represented by the ρ-term

$$
\begin{array}{rcl}
R & \triangleq & \{next(s1) \to s3, \ldots, next(s5) \to s5, \\
& & final(s1) \to false, \ldots, final(s5) \to true, \\
& & closed(s1) \to false, \ldots, closed(s5) \to true\}
\end{array}
$$

and we note $RC = R \cup B$.

The rules labeled with `double_f` and `double_c` are represented by the ρ-rules

$$
\begin{array}{rcl}
double_f(f) & \triangleq & x \to [im(RC)](\; [y \to [True \to next(y)]([im(RC)](not\ final(y)))] \\
& & \qquad\qquad\qquad ([[f](follow)]([im(RC)](x))))
\end{array}
$$

and respectively

$$
\begin{array}{rcl}
double_c(f) & \triangleq & x \to [im(RC)](\; [y \to [True \to next(y)]([im(RC)](not\ closed(y)))] \\
& & \qquad\qquad\qquad ([[f](follow)]([im(RC)](x))))
\end{array}
$$

The strategies from the module `automaton` are represented by the ρ-terms

$$
\begin{array}{lcl}
follow & \triangleq & follow \to \{s1 \to s2, s1 \to s3, s2 \to s5, s3 \to s2, s3 \to s4, s4 \to s1\} \\
gen_double(f) & \triangleq & gen_double \to [f](follow); [f](follow) \\
cond_double(f) & \triangleq & cond_double \to \{double_f(f), double_c(f)\}
\end{array}
$$

and we obtain the term representing the ELAN program `automaton`

$$
automaton \;\triangleq\; [\Theta](S)
$$

where

$$
S \;\triangleq\; f \to (y \to [\{follow, gen_double(f), cond_double(f)\}](y))
$$

The execution of the program `automaton` for evaluating the term `s1` with the strategy `cond_double` corresponds to the reduction of the term

$$
[[automaton](cond_double)](s1)
$$

In ELAN, we obtain for such an execution the results `2` and `5` and the reduction of the corresponding ρ-term leads to the set $\{2, 5\}$.

In Example 7.10 we presented a relatively simple ELAN module but, representative for the main features of the ELAN language. Following the same methodology, more complicated rules and strategies can be handled.

Notice that this provides, in particular, a very precise description of all the rewriting primitives, including the semantics of the conditional rewriting used by the language. To the best of our knowledge, this is the first *explicit* and *full* description of a rewrite based programming language.

8 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 manner the combination of algebraic and higher-order frameworks.

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 $\rho_\emptyset$-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 $\rho_\emptyset$-calculus is both conceptually simple as well as quite expressive. This allows us to represent the terms and reductions from λ-calculus and rewriting.

Using the ρ^{1st}-calculus, an extension of the ρ-calculus, appropriate definitions for term traversal operators and of a fixed-point operator can be given. This enables us to apply repeatedly a (set of) rewrite rule(s) and consequently to define a ρ-term representing the normalization according to a set of rewrite rules. Starting from this representation we showed how the ρ_T-calculus can be used to define conditional rewriting and to give a semantics to ELAN modules. Of course, this could be applied to many other frameworks, including rewrite based languages like ASF+SDF, ML, Maude or CafeOBJ but also production systems and non-deterministic transition systems.

Starting from these first results on the rewriting calculus, we have already explored, in subsequent papers, two different directions: the ρ-calculus with explicit substitutions and typed rewriting calculi. In [Cir00] we have proposed a version of the calculus where the substitution application is described at the same level as the other evaluation rules. Starting from the λ-calculus with explicit substitutions, and in particular the $\lambda\sigma_\Uparrow$-calculus, we developed the ρ-calculus with explicit substitutions, called the $\rho\sigma$-calculus and we showed that the $\rho\sigma$-calculus is confluent under the same conditions as the $\rho_\emptyset$-calculus.

The ρ-calculus is not terminating in the untyped case. In order to recover this property we have imposed in [CK00] a more strict discipline on the ρ-term formation by introducing a type for each term. We presented a type system for the $\rho_\emptyset$-calculus and we showed that it has the subject reduction and strong normalization properties. *i.e.* that the reduction of any well-typed term is terminating and preserves the type of the initial term. Additionally, we have given a new presentation *à la Church* to the ρ-calculus [CKL00b], together with nine (8+1) type systems which can be placed in a *ρ-cube* that extends the λ-cube of Barendregt. Quite interestingly, this typed calculus uses only one abstractor, namely the rule arrow.

We used the sets to represent the non-determinism and we mentioned that other structures can be used. For example, if we want to represent all the results of an application and not only the different results, then multisets must be used and if the order of the results is significant, then a list structure is more suitable. We have thus started the study of another description of the ρ-calculus having as parameter not only the matching theory but also the structure used for the results and we already showed its expressive power [CKL00a]. More precisely, we analyzed the correspondence between the ρ-calculus and two object oriented calculi: the *"Object Calculus"*

of Abadi and Cardelli [AC96] and the *"Lambda Calculus of Objects"* of Fisher, Honsell and Mitchell [FHM94]. The approach that we proposed allows the representation of objects in the style of the two mentioned calculi but also of more elaborate objects whose behavior is described by using the matching power.

As a new emergent framework, the ρ_T-calculus offers an original view point on rewriting and higher-order logic but also needs to further understand related topics. First, to go further in the study and the use of the ρ_T-calculus for the combination of first-order and higher-order paradigms, the investigation of the relationship of this calculus with higher-order rewrite concepts like CRS and HOR [vOvR93] should be deepened. Second, several directions should be investigated, amongst them, we can mention the following:

- The analysis of the properties of the ρ_T-calculus with a matching theory T more elaborate than syntactic matching.
- A generic description of the conditions which must be imposed for the matching theory T in order to obtain the confluence and the termination of the ρ_T-calculus should be defined and then, show that these conditions are satisfied for particular theories such as associativity and commutativity.
- The models of the rewriting calculus should be studied and compared with the ones of the algebraic as well as higher-order structures.
- As mentioned previously, we conjecture that the ρ^{1st}-calculus can not be expressed in the ρ-calculus because of the semantics of the empty set as rule application failure.

Finally, from the practical point of view, the various instances of the ρ-calculus must be further implemented and used as rewriting tools. We have already realized an implementation in ELAN of the $\rho_\emptyset$-calculus and we experimented with various evaluation strategies. This implementation could be further used in order to define object oriented paradigms. Dually, an object oriented version of the ELAN language has been realized [DK00], with a semantics given by the rewriting calculus.

This shows that this new calculus is very attractive in terms of semantics as well as unifying capabilities and we hope that it can serve in the future as a basic tool for the integration of semantic and logical frameworks.

Acknowledgements

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.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer Verlag, 1996.

[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.

[BKK+96] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[BKK98] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific. Also report LORIA 98-R-165.

[BKKM99] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. *ELAN* from the rewriting logic point of view. Research report, LORIA, November 1999.

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

[BR95] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

[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.

[Cas98] C. Castro. *Une approche déductive de la résolution de problèmes de satisfaction de contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France, 1998.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. 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.

[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.

[Cir99] H. Cirstea. Specifying authentication protocols using ELAN. In *Workshop on Modelling and Verification*, Besancon, France, December 1999.

[Cir00] H. Cirstea. *Calcul de Réécriture : Fondements et Applications.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.

[CK97] H. Cirstea and C. Kirchner. Theorem proving using computational systems: The case of the B predicate prover. In *Workshop CCL'97*, Schloβ Dagstuhl, Germany, September 1997.

[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.

[CK00] H. Cirstea and C. Kirchner. The simply typed rewriting calculus. In *3rd International Workshop on Rewriting Logic and its Applications*, Kanazawa (Japan), September 2000.

[CKL00a] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. October 2000. Submitted.

[CKL00b] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. November 2000. Submitted.

[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.

[Deu96] A. Deursen. An Overview of ASF+SDF. In *Language Prototyping*, pages 1–31. World Scientific, 1996. ISBN 981-02-2732-9.

[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.

[DK00] H. Dubois and H. Kirchner. Objects, rules and strategies in ELAN. In *Proceedings of the second AMAST workshop on Algebraic Methods in Language Processing, Iowa City, Iowa, USA*, May 2000.

[DO90] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.

[Dow92] G. Dowek. Third order matching is decidable. In *Proceedings of LICS'92*, Santa-Cruz (California, USA), June 1992.

[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.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specializatio n. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[FN97] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.

[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.

[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.

[Kli93] P. Klint. The ASF+SDF Meta-environment User's Guide. Technical report, CWI, 1993.

[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.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[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.

[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.

[Pro00] Protheo Team. The ELAN home page. WWW Page, 2000. `http://www.loria.fr/ELAN`.

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

[Tur37] A. M. Turing. The $\wp$-functions in λ-K-conversion. *The Journal of Symbolic Logic*, 2:164, 1937.

[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.

[VeAB98] E. Visser and Z. el Abidine Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume16.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[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.

[vOvR93] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In *HOA'93*, volume 816 of *Lecture Notes in Computer Science*, pages 276–304. Springer-Verlag, 1993.

[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.

Contents

1 Introduction 1
 1.1 Rewriting, computer science and logic 1
 1.2 How does the rewriting calculus work? 2
 1.3 Rewriting relation versus rewriting calculus 3
 1.4 Integration of first-order rewriting and higher-order logic 4
 1.5 Basic properties and uses of the ρ-calculus 4
 1.6 Structure of this paper 5
2 Definition of the ρ_T-calculus 5
 2.1 Syntax of the ρ_T-calculus 6
 2.2 Grafting versus substitution 8
 2.3 Matching 9
 2.4 Evaluation rules of the ρ_T-calculus 11
  2.4.1 Applying rewrite rules 11
  2.4.2 Applying operators 12
  2.4.3 Handling sets in the ρ_T-calculus 13
  2.4.4 Flattening sets in the ρ_T-calculus 14
  2.4.5 Using the ρ_T-calculus 15
 2.5 Evaluation strategies for the ρ_T-calculus 17
 2.6 Summary 17
3 The $\rho_\emptyset$-calculus 18
 3.1 The raw $\rho_\emptyset$-calculus is not confluent 19
 3.2 Enforcing confluence using strategies 22
4 Encoding λ-calculus and term rewriting in the $\rho_\emptyset$-calculus 28
 4.1 Encoding the λ-calculus 28
 4.2 Encoding rewriting 31
5 Recursion and term traversal operators 33
 5.1 Some auxiliary operators 34
 5.2 The $first$ operator 34
 5.3 Term traversal operators 36
 5.4 Iterators 39
  5.4.1 Multiple applications 40
  5.4.2 Singular applications 43
 5.5 Repetition and normalization operators 44
6 Using the ρ^{1st}-calculus 47
 6.1 Encoding rewriting in the ρ^{1st}-calculus 47
 6.2 Encoding conditional rewriting 48

6.2.1 Definition of conditional rewriting 48
6.2.2 Encoding 48
7 The rewriting calculus as a semantics of ELAN 51
7.1 ELAN's rewrite rules 51
7.2 The ρ-calculus representation of ELAN rules 53
7.2.1 Rules with local assignments 53
7.2.2 General strategies in the local assignments . . . 57
7.2.3 ELAN strategies and programs 57
8 Conclusion 62