INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Introduction to the rewriting calculus

Horatiu Cirstea , Claude Kirchner

N˚3818

Decembre 1999

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--3822--FR+ENG

INRIA
LORRAINE

Introduction to the rewriting calculus

Horatiu Cirstea , Claude Kirchner

Thème 2 — Génie logiciel
et calcul symbolique
Projet PROTHEO

Rapport de recherche n ° 3818 — Decembre 1999 — 50 pages

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.

Key-words: Rewriting, Strategy, Simplification, Reduction, Deduction, Matching.

(Résumé : tsvp)

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ

Introduction au calcul de réécriture

Résumé : Le ρ-calcul est un nouveau calcul qui intègre dans un cadre simple et uniforme la réécriture de premier ordre, le λ-calcul et les calculs non-déterministes. Ce rapport décrit le calcul depuis sa syntaxe jusqu'à ses propriétés de base dans le cas non typé.

Nous démontrons que le λ-calcul et la réécriture sont des cas particuliers du ρ-calcul dans le sens où la syntaxe et les règles d'inférence du ρ-calcul peuvent être restreintes afin d'obtenir les deux autres calculs. Finalement, nous utilisons le ρ-calcul pour donner une sémantique opérationnelle à ELAN, un langage basé sur la réécriture contrôlée par des stratégies.

Mots-clé : Réécriture, Stratégie, Simplification, Réduction,Déduction, Filtrage.

Contents

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 like in Mathematica [Wol99] 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 (say $a \to b$) to a term (like the constant a) as the object $[a \to b](a)$ which evaluates to the singleton $\{b\}$. This means that the rule application symbol $@$ (where @ is our notation for the placeholder) is part of the calculus syntax.

But the application of a rewrite rule may fail like in $[a \to b](c)$ that evaluates to the empty set $\emptyset$ or it can be reduced to a set with more than one element like exemplified later in this section and explained in Section 2.4. Of course, variables may be used in rewrite rules like in $[f(x) \to x](f(a))$. In this last case the evaluation mechanism of the calculus will reduce the application to $\{a\}$. In fact, when evaluating this expression, the variable x is bound to a 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 arrow operator, is also part of the calculus syntax. This is a powerful abstractor 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 $x \to t$ on 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 $Pair(x, y) \to x$, where $Pair$ is the function symbol that corresponds to the symbol $PAIR$, and the 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 like $\{x/u\}$ 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 like $f(x) \to g(x,y)$ that when applied to the term $f(a)$ (denoted by $[f(x) \to g(x,y)](f(a))$) evaluates to $\{g(a,y)\}$, leaving the variable y free. It may also rewrite an object into a rewrite rule like 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 $a \to b$ on 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 $a \to a$ on 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. That 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 ρ-calculus for deduction modulo purposes like 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 relation contains $(a, f(a))$, $(a, f(f(a)))$, $(f(a), f(f(a)))$, ... 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 like in $[a \to a]([a \to a]([a \to a](a)))$ that could be used as a ρ-representation 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 simple type 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] 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 abstractor, 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 simple type theory.

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 reductions results is the main source of non-confluence. Unlike in the standard definition of a rewrite step where the rule application yields always a result, in ρ-calculus a rule application always yields a unique result that can be a set with several elements, representing the non-deterministic choice of the corresponding results from rewriting, or with no elements ($\emptyset$), representing the failure. Therefore, the relation generated by the evaluation rules of the ρ-calculus is finer and consequently non-confluent.

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 [CK99b].

We will see that the ρ-calculus can be used for representing some simpler calculi like 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. 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. We then show in Section 4 how ρ-calculus can be used to encode in a uniform way term rewriting and λ-calculus. The encoding of 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 like 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] 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 Description 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* on 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:

- $\mathcal{T}(\mathcal{F}, \mathcal{X})$ 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 (if $n = 0$ we have the ρ-term $\emptyset$),
- if t and u are ρ-terms then $[t](u)$ is a ρ-term (application of the ρ-term t to the ρ-term u),
- if t and u are ρ-terms then $t \to u$ is a ρ-term (rewrite rule formation or abstraction).

We say that a ρ-term is *set-free* if it does not contain any set operator symbol (i.e. set braces or $\emptyset$). We call *functional position* of a ρ-term t, any occurrence p of the term whose function symbol belongs to $\mathcal{F}$, i.e. $t(p) \in \mathcal{F}$.

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 $f \in \mathcal{F}$.

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 allow 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 usual that the comma is associative, commutative and idempotent.

The main intuition behind this syntax is that a rewrite rule is an abstractor, the left-hand-side of which determines the bound variables and some contextual structure. 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 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.

Example 2.1 If we consider $\mathcal{F}_0 = \{a, b, c\}$, $\mathcal{F}_1 = \{f, g\}$, $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1$ 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 the evaluation of this application is $\{b\}$.
- $[a \to a](a)$; this denotes the application of the rewrite rule $a \to a$ to the term a. The evaluation of this application is $\{a\}$.
- $[f(x, y) \to g(x, y)](f(a, b))$; a classical rewrite rule application.
- $[x \to x + y](a)$; a rewrite rule with a free variable y and 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.
- $[[(x \to x + 1) \to (1 \to x)](a \to a + 1)](1)$; a more complicated ρ-term without corresponding standard rewrite rule or λ-term.

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

2.2 Grafting versus substitution

As for any calculus involving binders like the λ-calculus, α-conversion should be used in order to obtain a correct substitution calculus and the first-order substitution (called here grafting) is not directly suitable for 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 2.3. It computes a variant of a ρ-term which is equivalent modulo α-conversion to the initial term.

Definition 2.2 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 2.3 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 2.4 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 two notions of substitution and grafting as follow:

- 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 $(x, u) \in \theta$	– $\bar{\Theta}(x) = u$, if $(x, u) \in \theta$
– $\Theta(\{t\}) = \{\Theta(t)\}$	– $\bar{\Theta}(\{t\}) = \{\bar{\Theta}(t)\}$
– $\Theta(f(u_1, \ldots, u_n)) = (f(\Theta(u_1), \ldots, \Theta(u_n)))$	– $\bar{\Theta}(f(u_1, \ldots, u_n)) = (f(\bar{\Theta}(u_1), \ldots, \bar{\Theta}(u_n)))$
– $\Theta([t](u)) = [\Theta(t)](\Theta(u))$	– $\bar{\Theta}([t](u)) = [\bar{\Theta}(t)](\bar{\Theta}(u))$
– $\Theta(u \to v) = \Theta(u') \to \Theta(v')$	– $\bar{\Theta}(u \to v) = \bar{\Theta}(u) \to \bar{\Theta}(v)$

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:
$u' = \{x_i \mapsto z_i\}_{x_i \in FV(u)}\ \alpha_{FV(u) \cup \mathcal{V}ar(\theta)}(u)$,
$v' = \{x_i \mapsto z_i\}_{x_i \in FV(u)}\ \alpha_{FV(u) \cup \mathcal{V}ar(\theta)}(v)$.
The set of variables $\{x_1, \ldots, x_n\}$ is called the domain of the substitution Θ or of the grafting $\bar{\Theta}$ and is denoted by $\mathcal{D}om(\Theta)$ or $\mathcal{D}om(\bar{\Theta})$ respectively.

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

There is nothing new in the definition of substitution and grafting except that the abstraction works here on terms and not only on variables. The burden of variable handling could be avoided by using an explicit substitution mechanism in the spirit of [CHL96]. We sketched such an approach in [CK99a] and this is detailed in the forthcoming [CK99c].

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 2.5 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 at using constraints as in constrained higher-order resolution [Hue73] or constrained deduction [KKR90b]. 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, Rin94].

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

Figure 1: *SyntacticMatching* - Rules for syntactic matching

Proposition 2.1 [KK98] 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 $\mathcal{S}olution(t \ll^?_\emptyset t) = \{\mathbb{ID}\}$.

Example 2.2 If we consider the matching problem $(f(x, g(x, y)) \ll^?_\emptyset f(a, g(a, b))$, 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 $(x \ll^?_\emptyset a) \wedge (x \ll^?_\emptyset a) \wedge (y \ll^?_\emptyset b)$ and $\mathcal{S}olution(f(x, g(x, y)) \ll^?_\emptyset f(a, g(a, b)) = \{\{x/a, y/b\}\}$.

For the matching problem $(f(x, x) \ll^?_\emptyset f(a, b))$ 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, $\mathcal{S}olution(f(x, x) \ll^?_\emptyset f(a, b)) = \emptyset$.

This syntactic matching algorithm can be extended in a natural way when a symbol $+$ is assumed to be commutative. In this case, the previous set of rules should be completed with:

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

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 2.3 When matching modulo commutativity a term like $x+y$, with $+$ defined as commutative, against the term $a+b$, the rule $CommDec$ leads to

$$((x \ll^? a \wedge y \ll^? b) \vee (x \ll^? b \wedge y \ll^? a))$$

and thus, $\mathcal{S}olution(x+y \ll^? a+b) = \{\{x/a, y/b\}, \{x/b, y/a\}\}$.

Matching modulo associativity-commutativity (AC) is often used. It could be defined either in a rule based way like in [AK92, KR98] or in a semantic way like in [Eke93]. 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

We assume given a theory T over ρ-terms having a decidable matching problem. The use of constraints would allow to drop this last restriction, but we left it out of the scope of this paper.

2.4.1 Handling applications in the ρ_T-calculus

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 specificity they are described in Figure 2 to 5.

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.

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

Figure 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 p (i.e. $\mathcal{S}olution(l \ll^?_T p)$). As mentioned above, in the general case, the matching is not unitary and thus we should deal with (empty or infinite) sets of substitutions. We consider a substitution application at the meta-level of the calculus represented by the operator "@⟪@⟫" (where @ represents the placeholder as in the ELAN syntax) whose behavior is described by the meta-rule $Propagate$:

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

The result of the application of a set of substitutions $\{\sigma_1, \ldots, \sigma_n, \ldots\}$ on 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 on the term r as detailed above. Notice that when n is 0, i.e. the set of substitutions is empty, the resulting set of instantiated terms is also empty. 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, like 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 (like in the empty theory), a finite set (like 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 on t in the sense of a matching failure between l and t. We will see that this has important consequences concerning the behavior of the calculus. The first important one is that when evaluating a ρ-term, the number of set braces introduced counts the number of ($Fire$ and $Congruence$) evaluation steps performed in that derivation.

In order to push rewrite rule application deeper into terms, we introduce the two $Congruence$ evaluation rules of Figure 3. They deal with the application of a term of the form $f(u_1, \ldots, u_n)$ (where $f \in \mathcal{F}_n$) to another

term of a similar form. When we have the same head symbol for the two terms of the application $[u](v)$ the arguments of the term u are applied on those of the term v argument-wise. If the head symbols are not the same, an empty set is obtained.

$$\begin{array}{lll} Congruence & [f(u_1,\ldots,u_n)](f(v_1,\ldots,v_n)) & \Longrightarrow \{f([u_1](v_1),\ldots,[u_n](v_n))\} \\ Congruence\_fail & [f(u_1,\ldots,u_n)](g(v_1,\ldots,v_m)) & \Longrightarrow \emptyset \end{array}$$

Figure 3: The evaluation rules *Congruence* of the ρ_T-calculus

Remark: The *Congruence* rules are redundant with respect to *Fire*. Indeed, one could notice that the application of a term $f(u_1,\ldots,u_n)$ on another ρ-term t (i.e. the term $[f(u_1,\ldots,u_n)](t)$) evaluates, using the rules *Congruence* and $Congruence\_fail$, 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 term t (in a formal way, 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 a more concise representation of terms.

2.4.2 Handling sets in the ρ_T-calculus

The reductions corresponding to the cases where some sub-terms are sets are defined by the evaluation rules in Figure 4. These rules describe the propagation of the sets on the constructors of the ρ-terms: the rules *Distrib* and *Batch* for the application, $Switch_L$ and $Switch_R$ for the abstraction and *OpOnSet* for functions.

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

Figure 4: The evaluation rules *Set* of the ρ_T-calculus

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 $\emptyset$, the number of set symbols in a term counts the number of rules *Fire* and *Congruence* that have been applied for its evaluation. Due to the way the empty set is handled, the applications of the rules *Fire* and *Congruence* in the evaluation branches yielding to an $\emptyset$ result are not taken into account.

For example, let us consider a ρ-term t that evaluates in the ρ_T-calculus to the term $\{\{a\},\{c,\{d\}\}\}$ and assume that initially we had no set symbols in the term t. Then, we can say that we had *1 evaluation* step for the sub-term $\{a\}$ and *2 evaluation* steps for the sub-term $\{c,\{d\}\}$ and thus, *4 evaluation* steps for the final term. The *evaluation* considered in this example are just application of the evaluation rules *Fire* and *Congruence*.

A result of the form $\{\}$ represents a failure of a rule application and failures are *strictly* propagated in ρ-terms by the *Set* rules:

- for instance, the ρ-term $f(\emptyset,a)$ evaluates to $\emptyset$ using the rule *OpOnSet*.

- notice that in that case, the information on the number of *Fire* and *Congruence evaluation* steps is lost.

This behavior of the calculus could be summarized by stating that failure propagation by *Set* rules is strict on all operators but set. We will see later that *Fire* may induce non-strict propagations in some particular cases (see Example 3.5 on page 18).

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.

2.4.3 Flattening sets in the ρ_T-calculus

The evaluation rule that corresponds to the set propagation for set symbols, that would properly preserve the number of set braces, is the evaluation rule *Flat_one*:

$$\begin{array}{rl} \mathit{Flat\_one} & \{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\} \\ \Longrightarrow & \{\{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\}\} \quad \text{if } m \in \mathbf{N}^* \end{array}$$

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 use the evaluation rule *Elim* that eliminates the (redundant) set symbols:

$$\mathit{Elim} \quad \{\{u_1, \ldots, u_m\}\} \Longrightarrow \{u_1, \ldots, u_m\}$$

Another possibility is to flatten the sets and forget about the number of braces using the evaluation rule *Flat* in Figure 5 that combines the rules *Flat_one* 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.

$$\begin{array}{rl} \mathit{Flat} & \{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\} \\ \Longrightarrow & \{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\} \end{array}$$

Figure 5: The evaluation rules *Flat* of the ρ_T-calculus

This design decision to use sets to represent 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 set axioms. 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*, *Congruence* and *Congruence_fail* 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*, *Congruence*, *Congruence_fail*) and a set of *computation* rules (*Distrib*, *Batch*, Switch_L, Switch_R, *OpOnSet*, *Flat*) and that the reduction behaves as a in deduction modulo [DHK98]. This mean that we can consider the computation rules as describing a congruence modulo which the deduction rules are applied.

2.4.4 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 [Pro].

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. 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, it never fails and gives only one result.

There is no difficulty to represent more elaborated 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 $\rho_\emptyset$-calculus strictly embeds λ-calculus.

Now, if we introduce contextual information in the left-hand sides of the rewrite rules we obtain classical rewrite rules like $f(a) \to f(b)$ or $f(x) \to g(x)$. 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 while in the $\rho_\emptyset$-calculus the term $[f(a) \to f(b)](f(c))$ evaluates to $\emptyset$.

When the matching is done modulo an equational theory we obtain interesting behaviors. Take, for example, the list operator $\circ$ that appends two lists with elements of sort *Elem*. Any object of sort *Elem* represents a list consisting of this only object.

If we define the operator $\circ$ as right-associative, the rewrite rule taking the first part of a list can be written in the associative ρ_A-calculus $l \circ l' \to l$ and when applied to the list $a \circ b \circ c \circ d$ gives 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 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 of the way the term $a \circ b \circ c \circ d$ is parenthesized.

Let consider now 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. For 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 let us define it as associative-commutative and use the rewrite rule $x \circ x \circ L \to L$ 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 set. For instance, in the ρ_{AC}-calculus the application $[x \circ x \circ L \to L](a \circ b \circ c \circ a \circ d)$ evaluates to $\{b \circ c \circ d\}$: the search for the two equal elements is done thanks to associativity and commutativity.

Another facility is due to the use of sets for handling non-determinism. This allows us to easily express the non-deterministic application of a set of rewrite rules on 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 like in the following reduction: $[\{x \otimes y \to x, x \otimes y \to y\}](a \otimes b) \longrightarrow_{Distrib} \{[x \otimes y \to x](a \otimes b), [x \otimes y \to y](a \otimes b)\} \longrightarrow_{Fire} \{\{a\}, \{b\}\} \longrightarrow_{Flat} \{a, b\}$.

2.5 Evaluation strategies for the ρ_T-calculus

The strategy $\mathcal{S}$ guiding the application of the evaluation rules of the ρ_T-calculus could be crucial for obtaining good properties for the calculus. In a first stage, the main property analyzed is the confluence of the calculus 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 2.6 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 is not restrictive. The empty strategy just does not allow any reduction.

The reasons for the non-confluence of the general calculus are explained in Section 3 and a solution is proposed for obtaining a confluent calculus. The 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 general ρ_T-calculus.

Definition 2.7 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 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 on terms as defined in Section 2.2,
3. the theory T,
4. the set of evaluation rules $\mathcal{E}$: *Fire*, *Congruence*, *Congruence_fail*, *Distrib*, *Batch*, $Switch_L$, $Switch_R$, *OpOnSet*, *Flat*,
5. an evaluation strategy $\mathcal{S}$ that guides the application of the evaluation rules.

$\rho_T = (\varrho_{-}(\mathcal{F},\mathcal{X}), T, \mathcal{S})$ is the notation we use 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

Because it is already quite general and unless specifically mentioned, in the rest of this paper we restrict to the $\rho_\emptyset$-calculus.

3.1 Definition

We now define the $\rho_\emptyset$-calculus as the general ρ-calculus where the matching theory is restricted to first-order syntactic matching. As an instance of definition 2.7 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 on 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 3.1 With the exception of the last term, all the ρ-terms from Example 2.1 are $\rho_\emptyset$-terms.

The following remarks should be made with regards to the restrictions introduced in the $\rho_\emptyset$-calculus:

$$
\begin{array}{lll}
Fire & & [l \to r](t) \\
 & \Longrightarrow & r\langle\!\langle Solution(l \ll^{?}_{\emptyset} t)\rangle\!\rangle \\
 & & \\
Congruence & & [f(u_1,\ldots,u_n)](f(v_1,\ldots,v_n)) \\
 & \Longrightarrow & \{f([u_1](v_1),\ldots,[u_n](v_n))\} \\
 & & \\
Congruence\_fail & & [f(u_1,\ldots,u_n)](g(v_1,\ldots,v_m)) \\
 & \Longrightarrow & \emptyset \\
 & & \\
Distrib & & [\{u_1,\ldots,u_n\}](v) \\
 & \Longrightarrow & \{[u_1](v),\ldots,[u_n](v)\} \\
 & & \\
Batch & & [v](\{u_1,\ldots,u_n\}) \\
 & \Longrightarrow & \{[v](u_1),\ldots,[v](u_n)\} \\
 & & \\
Switch_R & & u \to \{v_1,\ldots,v_n\} \\
 & \Longrightarrow & \{u \to v_n,\ldots,u \to v_n\} \\
 & & \\
OpOnSet & & f(v_1,\ldots,\{u_1,\ldots,u_m\},\ldots,v_n) \\
 & \Longrightarrow & \{f(v_1,\ldots,u_1,\ldots,v_n),\ldots,f(v_1,\ldots,u_m,\ldots,v_n)\} \\
 & & \\
Flat & & \{u_1,\ldots,\{v_1,\ldots,v_n\},\ldots,u_m\} \\
 & \Longrightarrow & \{u_1,\ldots,v_1,\ldots,v_n,\ldots,u_m\}
\end{array}
$$

Figure 6: The evaluation rules of the $\rho_\emptyset$-calculus

- Since first-order syntactic matching is unitary, the meta-rule *Propagate* from Section 2.4.1 gives always as result the singleton $\{\sigma r\}$ or the empty set. Hence, the evaluation rule *Fire* can be replaced by the following simpler two rules:

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

- The evaluation rule $Switch_L$ can never be used in the $\rho_\emptyset$-calculus due to the restricted syntax imposed on ρ-terms.
- For a specific instance of the ρ_T-calculus, there is a strong relationship between the terms allowed on the left-hand side of the rule and the theory T. Intuitively, the theory T should be powerful enough to allow enough rule application. For instance, it seems more interesting to use a 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 side of rules.
- The term restrictions are made only on the left-hand side of rewrite rules and not on the right-hand side and this clearly allows more terms than in λ-calculus or in term rewriting.

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 [KKR90a], and will be studied in forthcoming works.

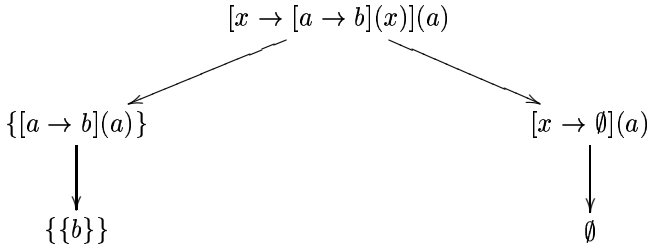
3.2 The raw $\rho_\emptyset$-calculus is not confluent

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 [CK99b] 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. If a matching failure is obtained due to a free variable of t and the matching succeeds when the variable is properly instantiated, then an example of the non-confluence is immediately obtained as shown in Example 3.2.

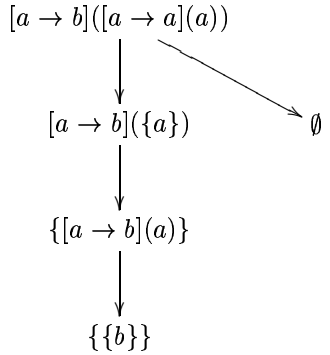
Example 3.2 [*Potentially instantiated variable*]



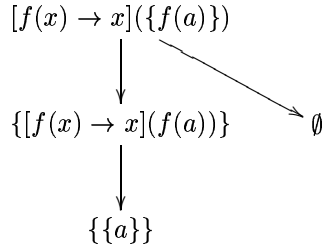
If a matching failure is obtained due to a non-reduced sub-term of t and the matching succeeds when the sub-term is reduced, then examples of the non-confluence can be easily found as presented in Example 3.3.a. A similar situation is obtained when the evaluation rule $Fire$ gives the $\emptyset$ result due to a matching failure but the application of another evaluation rule before the rule $Fire$ leads to a non-empty set like in Example 3.3.b.

Example 3.3 [*Under evaluated term*]

a.



b.

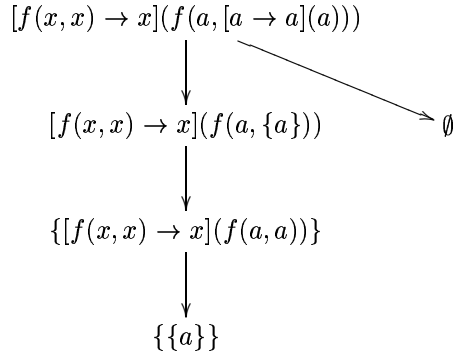


In order to avoid this kind of situation we should not allow the reduction of an application of the form $[l \to r](t)$ if the matching between the terms l and t fails due to the matching rules $SymbolVariableClash$ (Example 3.2) or $SymbolClash$ (Example 3.3.a, 3.3.b) and either some variables are not instantiated or some of the terms are not reduced or the term t is a set.

The matching rules $SymbolVariableClash$, $SymbolClash$ 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 3.2 where, in the term $[a \to b](x)$, a is a functional position and the corresponding position in the argument of the rewrite rule application is the variable position x. In Example 3.3.a and Example 3.3.b a functional position in the left-hand side of the rewrite rule corresponds to an abstraction and set position respectively and the condition is not satisfied.

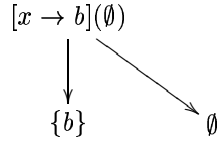
Therefore, we could consider that the evaluation rule $Fire$ is applied only when the condition on the functional positions is satisfied. Unfortunately, such a condition will not suffice for avoiding a non-appropriate matching fail due to the application of the rule $MergingClash$ if the term l is not linear, like shown in the next example:

Example 3.4 [*Non left linearity*]



Another pathological case arises when the term t contains an empty set or a sub-term 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 like in the next example:

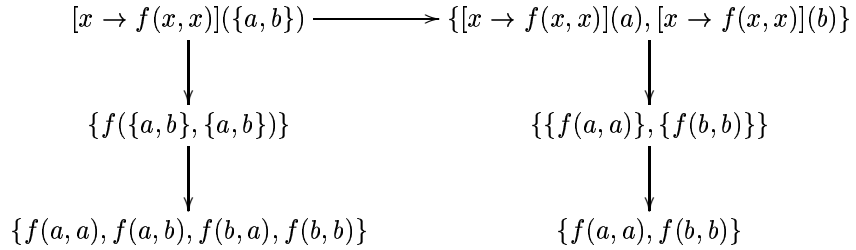
Example 3.5 [*Non strict failure propagation*]



We mention that a rewrite rule is *variable preserving* if the set of free variables of the left-hand is included in the set of free variables of the right-hand side, that is the free variables from the left-hand side are preserved in the right-hand side. In Section 3.3 we give a formal definition for the notion of *variable preserving* rewrite rule that takes into consideration all the operators of the ρ-calculus. The non-propagation of the failure is obtained when *non variable preserving* rewrite rules are applied on a term containing $\emptyset$. When a *variable preserving* rewrite rule is applied on a term containing $\emptyset$ the application result of substitutions of the form $\{x/\emptyset\}$ are not lost (as in Example 3.5) and thus the appropriate propagation of the $\emptyset$ is guaranteed.

When applying a non-right-linear rewrite rule on a term that contains sets with more than one element or terms that can be reduced to sets with more than one element we obtain undesirable results like in Example 3.6.

Example 3.6 [*Non right linearity*]



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

- the application of the rule *Fire* too early on an application containing non-instantiated variables,
- the application of the rule *Fire* too early on an application containing non-reduced terms,
- the application of the rule *Fire* too early on an application containing a non-left-linear rewrite rule,
- the rule *Fire* is used for reducing an application of a non-right-linear rewrite rule on a set containing more than one element,
- the rule *Fire* is used for reducing an application of a *non variable preserving* rewrite rule on a term containing the empty set.

3.3 Enforcing confluence using strategies

As we have seen the calculus is not confluent if no strategy is used to guide the application of the evaluation rules. But the confluence could be restored under an appropriate evaluation strategy. In particular this will impose the propagation of failure in terms to be strict and non-right-linear rewrite rules not to be applied on terms reducible to sets with more than one element.

A straightforward strategy that emerges from the above counterexamples consists in reducing an application by first pushing out the set braces, evaluating the argument of the rewrite rule and only when none of the previous reductions is possible, applying the evaluation rule *Fire*. This strategy 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 like the λ-calculus.

For a term u and $p_1, \ldots, p_n$ disjoint occurrences in u and $t_1, \ldots, t_n$ terms, we denote $u_{\lceil t_1 \rceil_{p_1} \cdots \lceil t_n \rceil_{p_n}}$ the term u with the term t_i at position p_i. The position of a sub-term into a set term is obtained by considering one of the tree representation of the set term. Also we should note that clearly set-freeness is not stable under the *Fire* and *Congruence* rules. This explain the technicalities of the next definition.

Definition 3.2 We say that an evaluation strategy whose reductions are denoted by $\longrightarrow_{\mathcal{SS}}$ is *set strict* if it satisfies the following conditions:

- for all ρ-terms u and t with u set-free, if $u_{\lceil t \rceil_p} \xrightarrow{*}_{\mathcal{SS}} u_{\lceil \emptyset \rceil_p}$ then for all term v such that $u_{\lceil t \rceil_p} \xrightarrow{*}_{\mathcal{SS}} v$, we have $v \xrightarrow{*}_{\mathcal{CS}} \emptyset$ (i.e. "error" propagation is strict on all operators except set formation),
- for all ρ-terms l, r, t, u and t' with u set-free, if $[l \to r](t) \xrightarrow{Fire}_{\mathcal{SS}} r_{\lceil u_{\lceil t' \rceil_i \lceil t' \rceil_j} \rceil_k}$ then for all ρ-term v such that $t' \xrightarrow{*}_{\mathcal{SS}} v$, there exists a ρ-term w such that $v \xrightarrow{*}_{\mathcal{SS}} \{w\}$ or $v \xrightarrow{*}_{\mathcal{SS}} \emptyset$ (i.e. t' is always reducible to a set with at most one element if it is involved in a non-linear context).

When restricting to the presentation of the λ-calculus in the ρ-calculus, like exemplified in the previous section and detailed later in Section 4.1, the matching never fails and always gives as result a singleton. Thus, the conditions from Definition 3.2 are trivially satisfied for any reduction and we can say that no specific strategy is needed to ensure the confluence of this specific instance of the ρ-calculus.

The confluence problems presented in Section 3.2 are avoided when a *set strict* evaluation strategy is used. More operational strategies should be provided for obtaining the confluence of the calculus and we present below two of them.

Definition 3.3 We call *ConfStrat* the class of strategies that consist in applying the rule *Fire* on a redex $[l \to r](t)$ at occurrence p in the term u (i.e. of the form $u_{\lceil [l \to r](t) \rceil_p}$) only when:

- either
  - l is linear and
  - the term t cannot be instantiated or reduced to an empty set and
  - the term t cannot be instantiated or reduced to a set with more than one element and
  - the set of functional positions of the term l is included in the set of functional positions of the term t
- or
  - l is linear and
  - r is linear on the variables of l and
  - the term t cannot be instantiated or reduced to an empty set and
  - the set of functional positions of the term l is included in the set of functional positions of the term t
- or
  - the term t contains no free variables bound in u, no redexes and no sets.

The conditions on the application of the rule *Fire* w.r.t. the number of elements of a set can be relaxed if we add some constraints on the structure of ρ-terms. First, we introduce a notion similar to that of free variable but that considers the behavior of the ρ-calculus symbols.

Definition 3.4 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 $PV(t) = \bigcap_{i=1,\ldots,n} PV(u_i)$,
3. if $t = f(u_1, \ldots, u_n)$ then $PV(t) = \bigcup_{i=1,\ldots,n} PV(u_i)$,
4. if $t = [u](v)$ then $PV(t) = PV(u) \cup PV(v)$,
5. if $t = u \to v$ then $PV(t) = PV(v) \setminus PV(u)$.

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 $PV(\{x, f(x, y)\}) = \{x\}$.

Definition 3.5 We say that the ρ-term $l \to r$ is *variable preserving* if $FV(l) \subseteq PV(r)$.

Example 3.7 The rewrite rule $x \to f(x, y)$ is *variable preserving* while the rewrite rule $x \to \{x, y\}$ is *non variable preserving.*

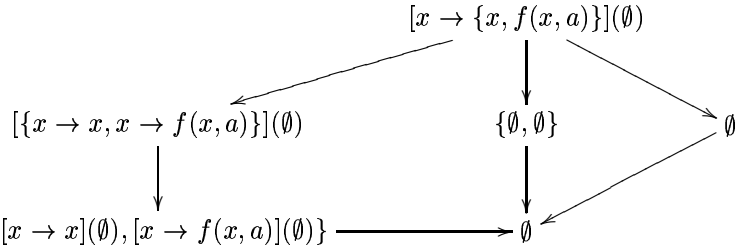
The rewrite rule $\{f(x), g(x)\} \to x$ is *variable preserving* while the rewrite rule $\{f(x), g(y)\} \to x$ is *non variable preserving.* If the definition of *variable preserving* rewrite rules had asked for the condition $PV(l) \subseteq PV(t)$ instead, then the second rewrite rule would have become *variable preserving* as well. This is not desirable since the rewrite rule $\{f(x), g(y)\} \to x$ reduces to $\{f(x) \to x, g(y) \to x\}$ and only the first one is *variable preserving.* Notice that these last rewrite rules contain sets in their left-hand side and thus are not proper terms of the $\rho_\emptyset$-calculus, but we gave them for a better understanding of the *variable preserving* notion.

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 3.5 can be changed to $\mathcal{V}ar(l) \subseteq PV(t)$.

Let us apply a *variable preserving* rewrite rule $l \to r$ on a term t, such that the matching substitution between l and t is the substitution σ. The application of the rewrite rule $l \to r$ on t gives as result the term $\{\sigma r\}$ and since $l \to r$ is *variable preserving* we have immediately $\mathcal{D}om(\sigma) \subseteq PV(r)$. Therefore, if $\emptyset$ is a sub-term of t, since it cannot appear in l, it consequently is in σ. So we are sure that $\emptyset$ is a sub-term of σr (since the rule is *variable preserving*) and thus we avoid non-confluent results like in Example 3.5.

Furthermore, if the right-hand side of a *variable preserving* rewrite rule is a set, since $\mathcal{D}om(\sigma) \subseteq PV(r)$ we deduce that $\mathcal{D}om(\sigma)$ is included in each of the elements of the set r. If $\emptyset$ is a sub-term of t and consequently is in σ, then $\emptyset$ is a sub-term of all the elements of the set r and thus the $\emptyset$ is strictly propagated in this case as well. The same reasoning can be done if r is not a set but some of its sub-terms are sets.

Example 3.8 A *variable preserving* rule applied on $\emptyset$ gives only one result:



while a *non variable preserving* one yields two different results as shown in Example 3.5.

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 *variable preserving* and thus the strict propagation of the empty set is ensured on all the right-hand sides of the obtained rewrite rules.

Definition 3.6 We call *ConfStratPreserve* the class of strategies that consist in applying the rule *Fire* on a redex $[l \to r](t)$ at occurrence p in the term u (i.e. of the form $u_{[[l \to r](t)]_p}$) only when:

- either

- l is linear and
- either $l \to r$ is *variable preserving* or t cannot be instantiated or reduced to an empty set and
- the term t cannot be instantiated or reduced to a set with more than one element and
- the set of functional positions of the term l is included in the set of functional positions of the term t

- or
  - l is linear and
  - r is linear on the variables of l and
  - either $l \to r$ is *variable preserving* or the term t cannot be instantiated or reduced to an empty set and
  - the set of functional positions of the term l is included in the set of functional positions of the term t

- or
  - the term t contains no free variables bound in u, no redexes and no sets.

Comparing to the definition of *ConfStrat*, we added the possibility to test either the *variable preserving* condition on the rewrite rule $l \to r$ or the condition on the reducibility of the term t to an empty set. If the rewrite rule $l \to r$ is *variable preserving*, the other conditions of the second alternative test only structural properties of the ρ-terms involved and does not depend on the possible reductions of the term t. If the rewrite rule $l \to r$ is *non variable preserving*, we still have to test the reducibility conditions that can be undecidable.

The condition that a set cannot be instantiated or reduced to a set with more than one element can be transformed into a structural but more restrictive condition. Hence, instead of testing the reducibility of the term t to a set with more than one element we can just test if the term t contains any free variables or sets with more than one element. For example, the ρ-term $[a \to \{c, d\}](d)$ contains a doubleton but it cannot be reduced to a doubleton and thus, depending on the condition we have used, the evaluation rule $Fire$ cannot or can be applied on the term $[x \to f(x, x)]([a \to \{c, d\}](d))$.

Proposition 3.1 [CK99b] When using an evaluation strategy from the class *ConfStrat* or the class *ConfStrat-Preserve*, the $\rho_\emptyset$-calculus is confluent.

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), like explained in Section 2.4, the overall confluence proof is different but uses lemmas similar to the ones of the former case. Therefore Proposition 3.1 can be extended to a ρ_E-calculus modulo a specific decidable and finitary equational matching E.

4 Encoding the 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, like 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 $\Lambda(\mathcal{X})$, the λ-calculus with names, 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:

$$P \xrightarrow{*}_\beta Q.$$

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

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

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

Definition 4.2 The ρ_λ-calculus is the ρ-calculus defined by:

- the $\mathcal{F}_\lambda$ terms,
- the (matching) theory $T = \emptyset$,
- 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. An immediate consequence of this restricted syntax is that the matching performed in the evaluation rule $Fire$ always succeeds and the solution of the matching equation that is necessarily of the form $x \ll^?_\emptyset t$ is always the singleton $\{x/t\}$.

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

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

Figure 7: The evaluation rules of the ρ_λ-calculus

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:

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

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:

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

The 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 3.3 are always satisfied. Therefore, the evaluation rule $Fire_\lambda$ can be used on any ρ_λ-application without loosing the confluence of the ρ_λ-calculus.

The restricted ρ_λ-calculus includes the λ-calculus modulo the notations for the application, abstraction and substitution application. The evaluation rule $Fire_\lambda$ initiates in the ρ-calculus (like the β-rule in the λ-calculus) the application of a substitution on a term. The rules *Congruence* are not used and the rules *Set* and *Flat* can be specialized for singletons and describe how to push out the set braces.

Reductions in the λ-calculus and in the ρ_λ-calculus are equivalent:

Proposition 4.1 Given t and t' two λ-terms. If $t \xrightarrow{*}_\beta t'$ then $\varphi(t) \xrightarrow{*}_{\rho_\lambda} \{\varphi(t')\}$ and for any ρ_λ-terms u and u', if $u \xrightarrow{*}_{\rho_\lambda} u'$ then $\delta(u) \xrightarrow{*}_\beta \delta(u')$.

Proof: It is enough to prove that if $t \longrightarrow_\beta t'$ then $\varphi(t) \xrightarrow{*}_{\rho_\lambda} \{\varphi(t')\}$ and afterwards extend the proof for any number of β reductions.

The proof is done by structural induction on the term t.

- If t is variable x, then $t' = x$ and $\varphi(t) = \varphi(t') = x$.
- If $t = \lambda x.u$ then $t' = \lambda x.u'$ with $u \xrightarrow{*}_\beta u'$ and $\varphi(t) = x \to \varphi(u)$. By induction, $\varphi(u) \xrightarrow{*}_{\rho_\lambda} \{\varphi(u')\}$, and thus
$$(x \to \varphi(u)) \xrightarrow{*}_{\rho_\lambda} (x \to \{\varphi(u')\}) \longrightarrow_{Switch_{R\lambda}} \{x \to \varphi(u')\}.$$
Since $\varphi(t') = (x \to \varphi(u'))$ the property is verified.
- If $t = (u\ v)$ then either $t' = (u'\ v')$ with $u \xrightarrow{*}_\beta u'$ and $v \xrightarrow{*}_\beta v'$ or $t = \lambda x.u\ v$ and $t' = \{x/v\}u$. In the first case $\varphi(t) = [\varphi(u)](\varphi(v))$ and by applying the induction we obtain
$$[\varphi(u)](\varphi(v)) \xrightarrow{*}_{\rho_\lambda} [\{\varphi(u')\}](\{\varphi(v')\}) \longrightarrow_{Distrib_\lambda, Batch_\lambda, Flat_\lambda} \{[\varphi(u')](\varphi(v'))\} = \{\varphi(t')\}.$$
For the second case $\varphi(t) = [x \to \varphi(u)](\varphi(v))$ and
$$[x \to \varphi(u)](\varphi(v)) \xrightarrow{*}_{\rho_\lambda} \{\{x/\varphi(v)\}\varphi(u)\}.$$
Since the application of a substitution is the same in the λ-calculus and in the ρ-calculus, we have, due to the definition of φ, $\varphi(t') = \varphi(\{x/v\}u) = \{x/\varphi(v)\}\varphi(u)$.

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

Hence, for the second implication, a corresponding reduction in the λ-calculus should be found only for the application of the evaluation rule $Fire_\lambda$. The proof in this case is very similar to the former one.

□

Example 4.1 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 $SKK = I$ corresponds a ρ_λ-reduction $[[S](K)](K) \xrightarrow{*}_{\rho_\lambda} \{I\}$.

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

As expected, we do not obtain the reduction $[[S](K)](K) \longrightarrow_\rho I$ but we can say that for any term t there exists a term u such that we have the two reductions $[[[S](K)](K)](t) \longrightarrow_\rho u$ and $[I](t) \longrightarrow_\rho u$. 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 on 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 add 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.

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 and rewriting logic, we just recall the basic notions that are consistent with [Mes89, DJ90, KKV95, BN98] to which the reader is referred for a more detailed presentation.

A *labeled rewrite theory* is a 5-tuple $\mathcal{R} = (\mathcal{X}, \mathcal{F}, E, \mathcal{L}, R)$ where $\mathcal{X}$ is a given countably infinite set of variables, $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols, E a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities, and R a set of labeled rewrite rules of the form $\ell : g \to d$ where $\ell \in \mathcal{L}$ and $g, d \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ satisfying $\mathcal{V}ar(d) \subseteq \mathcal{V}ar(g)$ and where the arity of ℓ is exactly the number of distinct variables in g. Note that the definitions can be extended to conditional rewriting at the cost of more technically complicated definitions. The reader is referred to [Mes92a] for a full treatment of this case.

A given labeled rewrite theory $\mathcal{R}$ entails the sequent $\pi : \langle t \rangle_A \to \langle t' \rangle_A$, which is denoted $\mathcal{R} \vdash \pi : \langle t \rangle_A \to \langle t' \rangle_A$ (or $\langle t \rangle_A \xrightarrow{*}_{\mathcal{R}} \langle t' \rangle_A$), if $\pi : \langle t \rangle_A \to \langle t' \rangle_A$ is obtained by some finite application of the deduction rules in Figure 8. Note that because of the rules *Reflexivity* and *Replacement*, if the rule $r = (\ell : g \to d) \in R$, then the sequent

$$
\begin{array}{ll}
\textit{Reflexivity} & \\
 & \Longrightarrow \\
 & \langle t \rangle_A : \langle t \rangle_A \to \langle t \rangle_A \\
 & \text{if } t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \\
\textit{Congruence} & \pi_1 : \langle t_1 \rangle_A \to \langle t'_1 \rangle_A, \ldots, \pi_n : \langle t_n \rangle_A \to \langle t'_n \rangle_A \\
 & \Longrightarrow \\
 & f(\pi_1, \ldots, \pi_n) : \langle f(t_1, \ldots, t_n) \rangle_A \to \langle f(t'_1, \ldots, t'_n) \rangle_A \\
 & \text{if } f \in \mathcal{F}_n \\
\textit{Replacement} & \pi_1 : \langle t_1 \rangle_A \to \langle t'_1 \rangle_A \ldots \pi_n : \langle t_n \rangle_A \to \langle t'_n \rangle_A \\
 & \Longrightarrow \\
 & \ell(\pi_1, \ldots, \pi_n) : \langle g(t_1, \ldots, t_n) \rangle_A \to \langle d(t'_1, \ldots, t'_n) \rangle_A \\
 & \text{if } \ell : g(x_1, \ldots, x_n) \to d(x_1, \ldots, x_n) \in R \\
\textit{Transitivity} & \pi_1 : \langle t_1 \rangle_A \to \langle t_2 \rangle_A,\ \pi_2 : \langle t_2 \rangle_A \to \langle t_3 \rangle_A \\
 & \Longrightarrow \\
 & \pi_1; \pi_2 : \langle t_1 \rangle_A \to \langle t_3 \rangle_A
\end{array}
$$

Figure 8: *REW*: The rules of unconditional rewrite deduction

$seq(r) = (\ell(\langle x_1 \rangle_A, \ldots, \langle x_n \rangle_A) : \langle g \rangle_A \to \langle d \rangle_A)$ can be derived. When we do not care about the labels, instead of a labelled rewrite theory, we simply speak of a *rewrite theory*.

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 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 by the ρ-operator $@$.

Since (matching) failure can take place in this case and sets with more elements are obtained when equational matching is not unitary, a reduction strategy like presented in Section 3.3 should be used in order to obtain the confluence of the resulting calculus.

For any reduction in a rewrite theory a corresponding reduction in the ρ-calculus can be found:

Proposition 4.2 Given t and t' two terms in $\mathcal{T}(\mathcal{F},\mathcal{X})$ and $\mathcal{R}$ a rewrite theory. If $t \xrightarrow{*}_{\mathcal{R}} t'$ then there exist ρ-terms $u_1,\ldots,u_n$ constructed using the rewrite rules in $\mathcal{R}$ such that $[u_n](\ldots[u_1](t)\ldots) \xrightarrow{*}_{\rho} \{t'\}$.

Proof: Let us consider a rewrite rule $(l \to r)$ that applies on the term t at position p and thus we have the reduction $t_{\lceil w\rceil_p} \longrightarrow_{\mathcal{R}} t_{\lceil \theta r\rceil_p} = t'$, where θ is the grafting such that $\theta l = w$.

Since $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ we can define a corresponding ρ-term $t_{\lceil l\to r\rceil_p}$ (i.e. t with $l \to r$ at occurrence p) and we obtain:

$$[t_{\lceil l\to r\rceil_p}](t_{\lceil w\rceil_p}) \xrightarrow{*}_{Congruence} \{t_{\lceil [l\to r](w)\rceil_p}\} \longrightarrow_{Fire} \{t_{\lceil \{\theta' r\}\rceil_p}\} \xrightarrow{*}_{OpOnSet}$$
$$\{\{t_{\lceil \theta' r\rceil_p}\}\} = \{\{t''\}\} \longrightarrow_{Flat} \{t''\}$$

where $\theta' \in Solution(l \ll^?_\emptyset w)$.

Since $\theta = \theta'$ and in this case grafting and substitution coincide (no abstraction is allowed in r) we obtain $t' = t''$.

When the rewrite rule $(l \to r)$ is applied at the top position of a term ($p = 0$) the corresponding ρ-term is, as expected, $[l \to r](w)$.

If several rewrite rules are applied in order to get the term t', the same procedure is used for each of the rewrite rules, and the final strategy is the composition of these rules. If we consider two rewrite rules s_1, s_2 applied in this order at positions p_1, p_2 on a term t we have the derivation:

$$t \longrightarrow_{s_1} t'' \longrightarrow_{s_2} t'$$

and the corresponding derivation in the ρ-calculus is:

$$[t''_{\lceil s_2\rceil_{p_2}}]([t_{\lceil s_1\rceil_{p_1}}](t)) \xrightarrow{*}_{\rho} [t''_{\lceil s_2\rceil_{p_2}}](\{t''\}) \xrightarrow{Batch}_{\rho} \{[t''_{\lceil s_2\rceil_{p_2}}](t'')\} \xrightarrow{*}_{\rho} \{\{t'\}\} \xrightarrow{Flat}_{\rho} \{t'\}.$$

If the derivation from t to t' consists of the rewrite rules $s_1,\ldots,s_n$ applied at positions $p_1,\ldots,p_n$ respectively, then the corresponding ρ-term is

$$[t^n_{\lceil s_n\rceil_{p_n}}](\ldots[t^2_{\lceil s_2\rceil_{p_2}}]([t^1_{\lceil s_1\rceil_{p_1}}](t))\ldots)$$

where t^{k+1} represents the term t after k steps of $\mathcal{R}$-reduction.

□

One can notice that the terms u_i used in the proof above are similar to the proof terms used in the labelled rewriting logic. 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 $[v;u](t)$, a notation that we will find again in the next section.

Until now we have used the evaluation rule *Congruence* for constructing the reduction that corresponds, in the ρ-calculus, to the reduction in the rewrite theory. As explained in Section 2.4, to any reduction performed using the rule *Congruence* corresponds a reduction that is done using the rule *Fire*. We consider the term u corresponding to a reduction in n (*Congruence*) steps: $[t^n_{\lceil s_n\rceil_{p_n}}](\ldots[t^2_{\lceil s_2\rceil_{p_2}}]([t^1_{\lceil s_1\rceil_{p_1}}](t))\ldots)$. The term u' that reduces to the same term as u but using *Fire* reductions is

$$[t^n_{\lceil x\rceil_{p_n}} \to t^n_{\lceil [s_n](x)\rceil_{p_n}}](\ldots([t^1_{\lceil x\rceil_{p_1}} \to t^1_{\lceil [s_1](x)\rceil_{p_1}}](t))\ldots).$$

Furthermore, if the rewrite rules s_i are of the form $l_i \to r_i$, $i = 1\ldots n$, the following term describes the same reduction:

$$[t^n_{\lceil l_n\rceil_{p_n}} \to t^n_{\lceil r_n\rceil_{p_n}}](\ldots([t^1_{\lceil l_1\rceil_{p_1}} \to t^1_{\lceil r_1\rceil_{p_1}}](t))\ldots).$$

Some simpler ρ-representation for a reduction in a rewrite system can be obtained if we focus on the application of only one rewrite rule. For the application of the rewrite rule $(l \to r)$ on the term t at position p

$$t_{\lceil w\rceil_p} \longrightarrow_{\mathcal{R}} t_{\lceil \theta r\rceil_p}$$

we can consider the ρ-reduction

$$t_{\lceil [l\to r](w)\rceil_p} \longrightarrow_{\rho} t_{\lceil \{\theta r\}\rceil_p} \longrightarrow_{\rho} \{t_{\lceil \theta r\rceil_p}\}.$$

This representation is obviously simpler and more intuitive but the systematic representation of an n-step rewriting reduction, $t \xrightarrow{*}_{\mathcal{R}} t'$, becomes more difficult in this case.

The last result shows that a derivation in term rewriting can be mimicked into an appropriate ρ-term that indeed represents the trace of the reduction. One can be much more powerful in ρ-calculus and describe the iteration of rule application by a ρ-term built using the fixed-point operators defined in the next section.

5 Recursion and term traversal operators

In Section 4.1 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 $\varsigma(u)$ that reduces to the term $\{v\}$. The lemma 4.2 provides a method for constructing a term $\varsigma(u)$ but only if we know all the reduction steps in the theory $\mathcal{R}$ from u to v.

We want to go further on and to give a method for constructing a term $\varsigma(u)$ 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 it 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 $\{\ldots, v, \ldots\}$?

In fact we would like to specialize not on any derivation but only on those leading to a normal form. This will allow us to get, in particular, a natural encoding of conditional term rewriting. Therefore, we want to answer the more specific question:

Given a rewrite theory $\mathcal{R}$ does it 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 $\{\ldots, v, \ldots\}$?

This means that we are looking for the description in the ρ-calculus of normalization strategies. This 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 representing such derivations at the object level. Of course, since the $\rho_\emptyset$-calculus contains the λ-calculus, any calculable function like the normalization one is expressible in the formalism. What we bring here, because of the "matching power" and 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 modulo 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* on 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 two 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 on 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 $x \to x$:

$$id \equiv x \to x.$$

The second 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 on the result of the application of u on t.

Therefore, we defined the operator ";" by:

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

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 9. We do not know currently how to express this operator in the basic ρ-calculus.

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

Figure 9: 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 parameters 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 is denoted by $\varrho^{1st}(\mathcal{F}, \mathcal{X})$.

We extend now the Definition 2.7 of the ρ^{1st}_T-calculus by considering the new operators and the corresponding evaluation rules presented in Figure 9.

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^{1st}_T-calculus (or rewriting 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 on terms as defined in Section 2.2,
- a theory T,
- the set of evaluation rules $\mathcal{E}_{\rho^{1st}}$: $Fire$, $Congruence$, $Congruence\_fail$, $Distrib$, $Batch$, $Switch_L$, $Switch_R$, $OpOnSet$, $Flat$, $First$, $First\_fail$, $First\_success$, $First\_single$,
- 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 5.1 $[first(a \to b, a \to c, a \to d)](a)$
$\longrightarrow_\rho \langle [a \to b](a), [a \to c](a), [a \to d](a)\rangle$
$\longrightarrow_\rho \langle \{b\}, [a \to c](a), [a \to d](a)\rangle$
$\longrightarrow_\rho \{\{b\}\}$
$\longrightarrow_\rho \{b\}$

Example 5.2 $[first(a \to b, b \to c, a \to d)](b)$
$\longrightarrow_\rho \langle [a \to b](b), [b \to c](b), [a \to d](b)\rangle$
$\longrightarrow_\rho \langle \emptyset, [b \to c](b), [a \to d](b)\rangle$
$\longrightarrow_\rho \langle [b \to c](b), [a \to d](b)\rangle$
$\longrightarrow_\rho \langle \{c\}, [a \to d](b)\rangle$
$\longrightarrow_\rho \langle \{c\}, [a \to d](b)\rangle$
$\longrightarrow_\rho \{\{c\}\}$
$\longrightarrow_\rho \{c\}$

Example 5.3 $[first(a \to b, a \to c, a \to d)](b)$
$\longrightarrow_\rho \langle [a \to b](b), [a \to c](b), [a \to d](b)\rangle$
$\longrightarrow_\rho \langle \emptyset, \emptyset, \emptyset\rangle$
$\longrightarrow_\rho \langle \emptyset, \emptyset\rangle$
$\longrightarrow_\rho \langle \emptyset\rangle$
$\longrightarrow_\rho \{\emptyset\}$
$\longrightarrow_\rho \emptyset$

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

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

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 *Congruence* but we want to define a generic operator that applies a ρ-term r on the sub-terms u_i, $i = 1 \ldots n$, of a term of the form $F(u_1, \ldots, u_n)$ 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 10. These operators are inspired by the operators of the *System S* described in [VeAB98].

$Traverse\_seq$	$[\Phi(r)](f(u_1, \ldots, u_n))$	$\Longrightarrow$	$\langle f([r](u_1), \ldots, u_n), \ldots, f(u_1, \ldots, [r](u_n))\rangle$
$Traverse\_seq_{const}$	$[\Phi(r)](c)$	$\Longrightarrow$	$\emptyset$
$Traverse\_par$	$[\Psi(r)](f(u_1, \ldots, u_n))$	$\Longrightarrow$	$\{f([r](u_1), \ldots, [r](u_n))\}$

Figure 10: The term traversal operators of the ρ_T-calculus

The application of the ρ-term $\Phi(r)$ on a term $t = f(u_1, \ldots, u_n)$ results in the application of the term r to one of the terms u_i, $i = 1, \ldots, n$ if there *exists* an i such that $[r](u_i)$ does not evaluate to $\emptyset$. More precisely, r is applied on the first u_i 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 on 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 $\Psi(r)$ on 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

$$\Phi'(r) \equiv first(f_1(r, id, \ldots, id), \ldots, f_1(id, \ldots, id, r), \ldots, f_m(r, id, \ldots, id), \ldots, f_m(id, \ldots, id, r))$$

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

with $c_i \in \mathcal{F}_0$, $i = 1, \ldots, n$, and $f_j \in \mathcal{F}_+$, $j = 1, \ldots, m$, it is easy to see that

$$[\Phi'(r)](f_k(u_1, \ldots, u_p)) \longrightarrow_\rho \{\langle \emptyset, \ldots, \emptyset, \{f_k([r](u_1), \ldots, u_p)\}, \ldots, \{f_k(u_1, \ldots, [r](u_p))\}, \emptyset, \ldots, \emptyset\rangle\}$$

and

$$[\Psi(r)](f_k(u_1, \ldots, u_p)) \longrightarrow_\rho \{f_k([r](u_1), \ldots, [r](u_p))\}.$$

When the traversal operators are applied to constants, the following reductions are obtained:

$$[\Phi'(r)](c_k) \longrightarrow_\rho \emptyset,$$

$$[\Psi(r)](c_k) \longrightarrow_\rho \{c_k\}.$$

The operator Φ' does not correspond exactly to the definition from the Figure 10 but a similar result is obtained when applying the terms $\Phi(r)$ and $\Phi'(r)$ to a term $f_k(u_1, \ldots, u_p)$.

If for all $i = 1, \ldots, p$ we have $[r](u_i) \longrightarrow_\rho \emptyset$ then, according to the definition from Figure 10, we obtain $[\Phi(r)](f_k(u_1, \ldots, u_p)) \longrightarrow_\rho \emptyset$. Otherwise, there exists an l such that the following evaluation is obtained: $[\Phi(r)](f_k(u_1, \ldots, u_p)) \longrightarrow_\rho \{f_k(u_1, \ldots, [r](u_l), \ldots, u_p)\}$.

According to the above definition of $\Phi'(r)$, the application $[\Phi'(r)](f_k(u_1, \ldots, u_p))$ evaluates to the ρ-term $\{\langle \emptyset, \ldots, \emptyset, \{f_k([r](u_1), \ldots, u_p)\}, \ldots, \{f_k(u_1, \ldots, [r](u_p))\}, \emptyset, \ldots, \emptyset\rangle\}$. This latter term is further evaluated to the term $\{\langle \{f_k([r](u_1), \ldots, u_p)\}, \ldots, \{f_k(u_1, \ldots, [r](u_p))\}, \emptyset, \ldots, \emptyset\rangle\}$. If for all $i = 1 \ldots p$ we have $[r](u_i) \longrightarrow_\rho \emptyset$ then, according to the definition from Figure 9, $[\Phi'(r)](f_k(u_1, \ldots, u_p)) \longrightarrow_\rho \emptyset$. Otherwise, there exists an l such that $[\Phi(r)](f_k(u_1, \ldots, u_p)) \longrightarrow_\rho \{\{f_k(u_1, \ldots, [r](u_l), \ldots, u_p)\}\}$ that reduces immediately to $\{f_k(u_1, \ldots, [r](u_l), \ldots, u_p)\}$.

So, we can state:

Lemma 5.1 The evaluations using the term traversal operators Φ and Ψ can be described in the ρ_T^{1st}-calculus.

For a more concise and clear presentation in the reductions described in this section we use the definition of the Φ operator as given in Figure 10.

5.4 Iterators

For the moment we have the possibility to apply a ρ-term to one or all the sub-terms of another ρ-term in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. If we want to get deeper in the structure of terms we should define some recursive operator that iterates the application of the operators Φ and Ψ and thus, pushes the application deeper into terms.

Using the fixed-point combinators of the λ-calculus we can define a ρ-term that applies recursively a given ρ-term. We use the classical fixed-point $\Theta = (A\ A)$ from the λ-calculus where

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

which is written in the ρ-calculus $\Theta = [A](A)$ with

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

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

$$\Theta\ G \longrightarrow_\beta G(\Theta\ G).$$

In ρ-calculus we have a similar reduction

$$[\Theta](G) \longrightarrow_\rho \{[G]([\Theta](G))\}$$

as this can be checked as follows:

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

We have obtained the desired result but the last application of the rule $Fire$ in the above evaluation is possible only if the confluence conditions presented in Definition 3.3 are satisfied. Hence, since the rewrite rule $y \to [y]([[A](A)](y))$ is not linear, we can apply the evaluation rule $Fire$ on the term $[y \to [y]([[A](A)](y))](G)$ only if the term G cannot be instantiated or reduced neither to an empty set nor to a set with more than one element. If the conditions on G are not satisfied the rule $Fire$ cannot be applied and all we can do is to reduce the term $[y \to [y]([[A](A)](y))](G)$ by reducing one of its sub-terms. We can thus reduce $[[A](A)](y) \equiv [\Theta](y)$ to the term $\{[y]([\Theta](y))\}$ and we obtain the term $[y \to [y](\{[y]([\Theta](y))\})](G)$. After several applications of the evaluation rules in order to push out the set symbols we get the term $[y \to [y]([y]([\Theta](y)))](G)$ that can be reduced only by reducing its sub-term $[\Theta](y)$ and thus the evaluation will never terminate.

Since the ρ-term $[\Theta](G)$ can obviously lead to infinite reductions even if the conditions on G are satisfied, a strategy should be imposed in order to avoid non-termination. In the evaluation of the ρ-term $[y \to [y]([[A](A)](y))](G) \equiv [y \to [y]([\Theta](y))](G)$ presented above if the term $[\Theta](y)$ was evaluated at each evaluation step the reduction would never end. Thus we should use a strategy that applies the evaluation rules on a term of the form $[\Theta](G)$ only when no other evaluation is possible. An operational strategy satisfying this condition should apply the evaluation rules first on the top position of terms and just when no evaluation rule is applicable on the top position evaluate the terms on deeper positions.

Now we have to define an appropriate G term that propagates the application of a ρ-term in the sub-terms of another ρ-term.

The first natural possibility is to define the ρ-term

$$GS_{sd}(r) \equiv f \to (x \to [\Psi(f); r](x)).$$

Let us consider the ρ-term

$$SpreadDownSingle(r) \equiv [\Theta](GS_{sd}(r))$$

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

$$
\begin{array}{ll}
 & [SpreadDownSingle(r)](t) \equiv [[\Theta](GS_{sd}(r))](t) \\
\longrightarrow_{\rho} & \{[[GS_{sd}(r)]([\Theta](GS_{sd}(r)))](t)\} \\
\equiv & \{[[GS_{sd}(r)](SpreadDownSingle(r))](t)\} \\
\equiv & \{[[f \to (x \to [\Psi(f); r](x))](SpreadDownSingle(r))](t)\} \\
\longrightarrow_{\rho} & \{[\{x \to [\Psi(SpreadDownSingle(r)); r](x)\}](t)\} \\
\longrightarrow_{\rho} & \{\{\{[\Psi(SpreadDownSingle(r)); r](f(t_1, \ldots, t_n))\}\}\} \\
\longrightarrow_{\rho} & \{\{\{\{[r](f([SpreadDownSingle(r)](t_1), \ldots, [SpreadDownSingle(r)](t_n)))\}\}\}\} \\
\longrightarrow_{\rho} & \{[r](f([SpreadDownSingle(r)](t_1), \ldots, [SpreadDownSingle(r)](t_n)))\}
\end{array}
$$

As we can see from this derivation the term $SpreadDownSingle(r)$ is applied recursively on the sub-terms of the term on which it is applied and the term r is applied on 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 result of the evaluation.

The evaluation presented above is possible only if the term $GS_{sd}(r)$ cannot be instantiated or reduced to an empty set or to a set with more than one element. This condition is obviously not respected if r is a set with more than one element. We want to prevent the evaluation of the term $GS_{sd}(r)$ to a set with more than one element even when r does not satisfy this condition. We define the term G_{sd}:

$$G_{sd}(r) \equiv f \to (x \to [first(\Psi(f); r, \emptyset)](x))$$

and

$$SpreadDown(r) \equiv [\Theta](G_{sd}(r)).$$

If $r = \{r_1, r_2\}$ the term $G_{sd}(r)$ evaluates as follows:

$$
\begin{array}{ll}
& f \to (x \to [first(\Psi(f); \{r_1, r_2\}, \emptyset)](x)) \\
\longrightarrow_\rho & f \to (x \to < [\Psi(f); \{r_1, r_2\}](x), [\emptyset](x) >) \\
\longrightarrow_\rho & f \to (x \to < [\{r_1, r_2\}]([\Psi(f)](x)), [\emptyset](x) >) \\
\longrightarrow_\rho & f \to (x \to < \{[r_1]([\Psi(f)](x)), [r_2]([\Psi(f)](x))\}, [\emptyset](x) >)
\end{array}
$$

Since this last term is not a set we obtain an evaluation similar to the previous approach even when r is a set with more than one element:

$$
\begin{array}{ll}
& [SpreadDown(r)](t) \equiv [[\Theta](G_{sd}(r))](t) \\
\longrightarrow_\rho & \{[[G_{sd}(r)]([\Theta](G_{sd}(r)))](t)\} \\
\equiv & \{[[G_{sd}(r)](SpreadDown(r))](t)\} \\
\equiv & \{[[f \to (x \to [first(\Psi(f); r, \emptyset)](x))](SpreadDown(r))](t)\} \\
\longrightarrow_\rho & \{[\{x \to [first(\Psi(SpreadDown(r)); r, \emptyset)](x)\}](t)\} \\
\longrightarrow_\rho & \{\{\{[first(\Psi(SpreadDown(r)); r, \emptyset)](f(t_1, \ldots, t_n))\}\}\} \\
\longrightarrow_\rho & \{\{\{< [first(\Psi(SpreadDown(r)); r](f(t_1, \ldots, t_n), [\emptyset](f(t_1, \ldots, t_n)) >\}\}\} \\
\longrightarrow_\rho & \{< [r](f([SpreadDown(r)](t_1), \ldots, [SpreadDown(r)](t_n))), [\emptyset](f(t_1, \ldots, t_n)) >\}
\end{array}
$$

that reduces to the same term as the term

$$\{[r](f([SpreadDown(r)](t_1), \ldots, [SpreadDown(r)](t_n)))\}.$$

One should notice that when we apply the operator *SpreadDown* on a constant we obtain the reduction:

$$[SpreadDown(r)](c) \longrightarrow_\rho \{[r](c)\}.$$

Example 5.4 If we use a strategy that applies the evaluation rules first on the top position the following derivation is obtained:

$$
\begin{array}{ll}
& [SpreadDown(\{a \to b, id\})](f(a, g(a))) \\
\longrightarrow_\rho & \{< [\{a \to b, id\}](f([SpreadDown(\{a \to b, id\})](a), [SpreadDown(\{a \to b, id\})](g(a)))), \\
& \quad [\emptyset](f(a, g(a))) >\} \\
\longrightarrow_\rho & \{< \{[a \to b](f([SpreadDown(\{a \to b, id\})](a), [SpreadDown(\{a \to b, id\})](g(a)))), \\
& \quad [id](f([SpreadDown(\{a \to b, id\})](a), [SpreadDown(\{a \to b, id\})](g(a)))\}, \\
& \quad [\emptyset](f(a, g(a))) >\} \\
\longrightarrow_\rho & \{< \{\{f([SpreadDown(\{a \to b, id\})](a), [SpreadDown(\{a \to b, id\})](g(a)))\}\}, \\
& \quad [\emptyset](f(a, g(a))) >\} \\
\longrightarrow_\rho & \{< \{f(\{[\{a \to b, id\}](a)\}, \{[\{a \to b, id\}](g([SpreadDown(\{a \to b, id\})](a))))\}, \\
& \quad [\emptyset](f(a, g(a))) >\} \\
\longrightarrow_\rho & \{< \{f(\{b, a\}, \{g([SpreadDown(\{a \to b, id\})](a))\})\}, [\emptyset](f(a, g(a))) >\} \\
\longrightarrow_\rho & \{< \{f(\{b, a\}, g(\{b, a\}))\}, [\emptyset](f(a, g(a))) >\} \\
\longrightarrow_\rho & \{f(b, g(b)), f(a, g(b)), f(b, g(a)), f(a, g(a))\}
\end{array}
$$

The operator $SpreadDown(r)$ looks similar to a bottom-up application of the term r but due to the strategy that should be imposed on the application of the evaluation rules this is not the case. In a bottom-up application the term

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

should have been reduced to

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

Example 5.5 Another terminating strategy would evaluate the terms of the form $[SpreadDown(r)](t)$ as soon as possible but delay the evaluation of the term $SpreadDown(r)$ as long as possible. In this case the operator $SpreadDown(r)$ corresponds to a bottom-up application of the term r.

Let us consider, for example, the term $[SpreadDown(\{f(b) \to b, a \to b\})](f(a))$ that leads to the evaluation of $\{[\{f(b) \to b, a \to b\}](f([SpreadDown(\{f(b) \to b, a \to b\})](a)))\}$. The term $[SpreadDown(\{f(b) \to b, a \to b\})](a)$ clearly evaluates to $\{b\}$ and thus we obtain the term $\{[\{f(b) \to b, a \to b\}](f(\{b\}))\}$ that that evaluates to $\{b\}$.

From an operational point of view this evaluation strategy is more difficult to implement because we have to detect the ρ-term *SpreadDown* during the derivation. If this term is explicit the detection is easy but when it is expanded with the corresponding definitions it becomes more difficult to detect it.

Our initial goal was to define a ρ-term that describes the propagation of a ρ-term in the sub-terms of another ρ-term. The disadvantage of the *SpreadDown* approach is that an application $[SpreadDown(r)](t)$ leads to an $\emptyset$ result as soon as the term r leads to an $\emptyset$ result when applied to a sub-term of t. If we want to leave unchanged the sub-terms on which the application of the term r evaluates to $\emptyset$ we use the term id like in Example 5.4 or like in the definition of the term G_{ssd} below that replaces the G_{sd} one:

$$G_{ssd}(r) \equiv f \to (x \to [first(\Psi(f), id); first(r, id)](x)).$$

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

$$G_{td}(r) \equiv f \to (x \to [first(r; \Psi(f), \emptyset)](x))$$

and we define the term

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

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:

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

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

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

As in the case of the operator *SpreadDown* the term

$$[Once_{bu}(r)](t) \equiv [[\Theta](H_{bu}(r))](t)$$

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 on the top position and only if this is not possible on deeper positions.

The following derivation illustrates the application of the operator $Once_{bu}(r)$ on a term t:

$$\begin{array}{ll}
 & [Once_{bu}(r)](t) \equiv [[\Theta](H_{bu}(r))](t) \\
\longrightarrow_\rho & \{[[H_{bu}(r)]([\Theta](H_{bu}(r)))](t)\} \\
\equiv & \{[[f \to (x \to [first(\Phi(f), r)](x))](Once_{bu}(r))](t)\} \\
\longrightarrow_\rho & \{[\{x \to [first(\Phi(Once_{bu}(r)), r)](x)\}](t)\} \\
\longrightarrow_\rho & \{\{\{[first(\Phi(Once_{bu}(r)), r)](f(t_1, \ldots, t_n))\}\}\} \\
\longrightarrow_\rho & \{\{\{\langle [\Phi(Once_{bu}(r))](f(t_1, \ldots, t_n)), [r](f(t_1, \ldots, t_n))\rangle\}\}\} \\
\longrightarrow_\rho & \{\langle\langle f([Once_{bu}(r)](t_1), \ldots, t_n), \ldots, f(t_1, \ldots, [Once_{bu}(r)](t_n))\rangle, [r](f(t_1, \ldots, t_n))\rangle\}
\end{array}$$

If all the terms $f(t_1, \ldots, [Once_{bu}(r)](t_k), \ldots, t_n)$, $k = 1, \ldots, n$, evaluate to the empty set and $[r](f(t_1, \ldots, t_n))$ evaluates to the empty set as well then the last term from the above reduction obviously evaluates to $\emptyset$.

If all the terms $f(t_1, \ldots, [Once_{bu}(r)](t_k), \ldots, t_n)$, $k = 1, \ldots, n$, evaluate to the empty set (the application of $Once_{bu}(r)$ on any sub-term t_i, $i = 1, \ldots, n$, of $f(t_1, \ldots, t_n)$ evaluates to $\emptyset$) but $[r](f(t_1, \ldots, t_n))$ does not evaluate to the empty set then the last term from the above reduction evaluates to

$$\{[r](f(t_1, \ldots, t_n))\}.$$

If at least one of the terms $f(t_1, \ldots, [Once_{bu}(r)](t_k), \ldots, t_n)$ does not evaluate to the empty set then the term $[r](f(t_1, \ldots, t_n))$ will be ignored later in the evaluation due to the evaluation rule $First\_success$ and the the result of the evaluation is the term

$$\{f(t_1, \ldots, [Once_{bu}(r)](t_k), \ldots, t_n)\}$$

such that for any $i < k$, $[Once_{bu}(r)](t_i) \longrightarrow_\rho \emptyset$ and $[Once_{bu}(r)](t_k)$ does not evaluate to $\emptyset$.

One should notice that when we apply the operator $Once_{bu}$ on a constant we obtain the reduction:

$$[Once_{bu}(r)](c) \longrightarrow_\rho \{[r](c)\}.$$

Hence, if p is the leftmost innermost position of $t_{\lceil u \rceil_p}$ where r can be applied successfully then the evaluation of the term $[Once_{bu}(r)](t)$ leads to the term $t_{\lceil [r](u) \rceil_p}$. If there exists no such position the result is $\emptyset$.

Example 5.6 The application of the rule $a \to b$ once on the term $f(a, g(a))$ is represented by the term $[Once_{bu}(a \to b)](f(a, g(a)))$ and the corresponding evaluation is presented below:

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

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:

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

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

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

In the application $[Once_{td}(r)](t)$ we apply first the term r at the top position and if it does not work we apply r deeper in the term t.

Lemma 5.2 The operators that apply once a term in a *bottom-up* or *top down* way ($Once_{bu}$, $Once_{td}$) can be expressed in the ρ_T^{1st}-calculus.

5.5 The *repeat* operator

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 we describe it by the evaluation rule from Figure 11.

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

Figure 11: The *repeat* operator

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

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

that is used for describing a *repeat* operator:

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

The following derivation is obtained when a $repeat(r)$ term is applied on a ρ-term t and the evaluation rules are guided by the same strategy as in the previous section:

$$\begin{array}{ll} & [repeat(r)](t) \equiv [[\Theta](I(r))](t) \\ \longrightarrow_\rho & \{[[I(r)]([\Theta](I(r)))](t)\} \\ \equiv & \{[[f \to (x \to [r; f](x))](repeat(r))](t)\} \\ \longrightarrow_\rho & \{[\{x \to [r; repeat(r)](x)\}](t)\} \\ \longrightarrow_\rho & \{\{\{[r; repeat(r)](t)\}\}\} \\ \longrightarrow_\rho & \{[repeat(r)]([r](t))\} \end{array}$$

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 employed. Second, when the evaluation terminates the result is always the empty set.

When the strategy applies the evaluation rules first on the top position of an application $[u](v)$ and only afterwards on the right sub-term v and then on the left sub-term u we do not obtain the desired result. In the reduction above the term $\{[repeat(r)]([r](t))\}$ is evaluated to $\{[repeat(r)]([r]([r](t)))\}$ and the reduction never ends.

The solution to this problem is to evaluate first the argument v of the application if the evaluation rules cannot be applied on the top position. In this case, if at some point in the evaluation the application of the term r evaluates to the empty set, then $\emptyset$ is strictly propagated and thus the term $[repeat(r)](t)$ evaluates to the empty set.

We still have the problem of obtaining always the $\emptyset$ result. 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 should not forget that we reduce an application $[u](v)$ by applying the evaluation rules on the top position, then on its argument v and only afterwards on the term u.

We modify the term $I(r)$ that becomes

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

and we define the term

$$repeat*\,(r) \equiv [\Theta](J(r))$$

that applied on a ρ-term t leads to the following derivation:

$$\begin{array}{ll} & [repeat*\,(r)](t) \equiv [[\Theta](J(r))](t) \\ \longrightarrow_\rho & \{[[J(r)]([\Theta](J(r)))](t)\} \\ \equiv & \{[[f \to (x \to [first(r; f, id)](x))](repeat*\,(r))](t)\} \\ \longrightarrow_\rho & \{[\{x \to [first(r; repeat*\,(r), id)](x)\}](t)\} \\ \longrightarrow_\rho & \{[first(r; repeat*\,(r), id)](t)\} \\ \longrightarrow_\rho & \{\langle [r; repeat*\,(r)](t), [id](t)\rangle\} \\ \longrightarrow_\rho & \{\langle [repeat*\,(r)]([r](t)), [id](t)\rangle\} \end{array}$$

If the term r applies successfully on t the evaluation is similar to the evaluation of the term

$$\{[repeat*\,(r)]([r](t))\} \longrightarrow_\rho \{[repeat*\,(r)](\{t'\})\} \longrightarrow_\rho \{[repeat*\,(r)](t')\}$$

and if $[r](t) \longrightarrow_\rho \emptyset$ then $[repeat*\,(r)]([r](t)) \longrightarrow_\rho \emptyset$ and the initial term evaluates to

$$\{[id](t)\} \longrightarrow_\rho \{t\}.$$

Notice that the result of $[repeat*\,(r)](t)$ represents the last term on which we have applied successfully the term r. This is the same as the initial term t only if r cannot be applied successfully even once.

Lemma 5.3 The operators that apply repeatedly a term until it is no more applicable (*repeat**) can be expressed in the ρ^{1st}_T-calculus.

Example 5.7 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:

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

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

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

and an *outermost* strategy is defined by

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

Corollary 5.1 The operators that normalize a term in a *bottom-up* or *top-down* way (im, om) 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 $\varsigma(u)$ described at the beginning of Section 5 is defined by

$$\varsigma(u) \equiv [im(\mathcal{R})](u)$$

or

$$\varsigma(u) \equiv [om(\mathcal{R})](u).$$

Example 5.8 If we denote by $\mathcal{R}$ the set of rewrite rules $\{a \to b, f(x, g(x)) \to x\}$ the following evaluation represents the innermost normalization of the term $f(a, g(a))$ according to the set of rules $\mathcal{R}$:

$$\begin{aligned}
& [im(\mathcal{R})](f(a,g(a))) \\
\equiv\ & [repeat*\,(Once_{bu}(\mathcal{R}))](f(a,g(a))) \\
\longrightarrow_\rho\ & \{\langle [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(a,g(a)))), [id](f(a,g(a)))\rangle\} \\
\longrightarrow_\rho\ & \{\langle [repeat*\,(Once_{bu}(\mathcal{R}))](\{f(b,g(a))\}), \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{[repeat*\,(Once_{bu}(\mathcal{R}))](f(b,g(a)))\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\langle [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(b,g(a))), [id](f(b,g(a)))\rangle\}\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\langle [repeat*\,(Once_{bu}(\mathcal{R}))](\{f(b,g(b))\}), \{f(b,g(a))\}\rangle\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\langle \{[repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(b,g(b)))\}, \{f(b,g(b))\}\rangle\}, \{f(b,g(a))\}\rangle\}, \\
& \qquad \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\langle \{[repeat*\,(Once_{bu}(\mathcal{R}))](\{b\})\}, \{f(b,g(b))\}\rangle\}, \{f(b,g(a))\}\rangle\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\langle \{\langle [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](b)), [id](b)\rangle, \{f(b,g(b))\}\rangle\}, \{f(b,g(a))\}\rangle\}, \\
& \qquad \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\langle \{\langle [repeat*\,(Once_{bu}(\mathcal{R}))](\emptyset), \{b\}\rangle, \{f(b,g(b))\}\rangle\}, \{f(b,g(a))\}\rangle\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\langle \{\langle \emptyset, \{b\}\rangle, \{f(b,g(b))\}\rangle\}, \{f(b,g(a))\}\rangle\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\langle \{\{\{b\}\}\}, \{f(b,g(b))\}\rangle\}, \{f(b,g(a))\}\rangle\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\{\{b\}\}, \{f(b,g(a))\}\rangle\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{\{\{b\}\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{\langle \{b\}, \{f(a,g(a))\}\rangle\} \\
\longrightarrow_\rho\ & \{b\}
\end{aligned}$$

If the rewrite theory $\mathcal{R}$ is not confluent the evaluation of the term $\varsigma(u)$ yields a set of results representing the possible results of the reduction of the term u in the rewrite theory $\mathcal{R}$.

In the following two examples we show the simplified reduction where the reductions

$$[repeat*\,(r)](t) \longrightarrow_\rho \{\langle [repeat*\,(r)]([r](t)), [id](t)\rangle\}$$

are replaced by

$$[repeat*\,(r)](t) \longrightarrow_\rho \{[repeat*\,(r)]([r](t))\}$$

if it is clear that $[r](t)$ does not evaluate to $\emptyset$.

Example 5.9 If we denote by $\mathcal{R}$ the set of rewrite rules $\{a \to b, a \to c, f(x, x) \to x\}$ the following simplified evaluation represents the innermost normalization of the term $f(a, a)$ according to the set of rules $\mathcal{R}$:

$$\begin{aligned}
& [im(\mathcal{R})](f(a,a)) \\
\equiv\ & [repeat*\,(Once_{bu}(\mathcal{R}))](f(a,a)) \\
\longrightarrow_\rho\ & \{[repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(a,a)))\} \\
\longrightarrow_\rho\ & \{[repeat*\,(Once_{bu}(\mathcal{R}))](\{f(\{b,c\},a)\})\} \\
\longrightarrow_\rho\ & \{[repeat*\,(Once_{bu}(\mathcal{R}))](f(\{b,c\},a))\} \\
\longrightarrow_\rho\ & \{[repeat*\,(Once_{bu}(\mathcal{R}))](\{f(b,a),f(c,a)\})\} \\
\longrightarrow_\rho\ & \{[repeat*\,(Once_{bu}(\mathcal{R}))](f(b,a)), [repeat*\,(Once_{bu}(\mathcal{R}))](f(c,a))\} \\
\longrightarrow_\rho\ & \{[repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(b,a))), \\
& \qquad [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(c,a)))\} \\
\longrightarrow_\rho\ & \{[repeat*\,(Once_{bu}(\mathcal{R}))](f(b,\{b,c\})), [repeat*\,(Once_{bu}(\mathcal{R}))](f(c,\{b,c\}))\} \\
\longrightarrow_\rho\ & \{[repeat*\,(Once_{bu}(\mathcal{R}))](f(b,b)), [repeat*\,(Once_{bu}(\mathcal{R}))](f(b,c)),
\end{aligned}$$

$$
\begin{array}{ll}
 & [repeat*\,(Once_{bu}(\mathcal{R}))](f(c,b)), [repeat*\,(Once_{bu}(\mathcal{R}))](f(c,c))\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(b,b))), \\
 & \quad [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(b,c))) \\
 & \quad [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(b,c))), \\
 & \quad [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](f(c,c)))\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{bu}(\mathcal{R}))](\{b\}), [repeat*\,(Once_{bu}(\mathcal{R}))](\{f(b,c)\}) \\
 & \quad [repeat*\,(Once_{bu}(\mathcal{R}))](\{f(c,b)\}), [repeat*\,(Once_{bu}(\mathcal{R}))](\{c\})\} \\
\longrightarrow_\rho & \{\{b\}, \{f(b,c)\}, \{f(c,b)\}, \{c\}\} \\
\longrightarrow_\rho & \{b, f(c,b), f(b,c), c\}
\end{array}
$$

Example 5.10 If we denote by $\mathcal{R}$ the set of rewrite rules $\{a \to b, a \to c, f(x,x) \to x\}$ the following simplified evaluation represents the outermost normalization of the term $f(a,a)$ according to the set of rules $\mathcal{R}$:

$$
\begin{array}{ll}
 & [om(\mathcal{R})](f(a,a)) \\
\equiv & [repeat*\,(Once_{td}(\mathcal{R}))](f(a,a)) \\
\longrightarrow_\rho & \{[repeat*\,(Once_{td}(\mathcal{R}))]([Once_{td}(\mathcal{R})](f(a,a)))\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{td}(\mathcal{R}))]([first(\mathcal{R}, \Phi(Once_{td}(\mathcal{R})))](f(a,a)))\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{td}(\mathcal{R}))](([\{a \to b, a \to c, f(x,x) \to x\}](f(a,a)), \\
 & \quad [\Phi(Once_{td}(\mathcal{R}))](f(a,a))))\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{td}(\mathcal{R}))](\{a\}, [\Phi(Once_{td}(\mathcal{R}))](f(a,a)))\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{td}(\mathcal{R}))](a)\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{td}(\mathcal{R}))]([Once_{td}(\mathcal{R})](a))\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{td}(\mathcal{R}))](\{b,c\})\} \\
\longrightarrow_\rho & \{[repeat*\,(Once_{td}(\mathcal{R}))](b), [repeat*\,(Once_{td}(\mathcal{R}))](c)\} \\
\longrightarrow_\rho & \{\langle [repeat*\,(Once_{td}(\mathcal{R}))]([Once_{td}(\mathcal{R})](b)), [id](b)\rangle, \\
 & \quad \langle [repeat*\,(Once_{td}(\mathcal{R}))]([Once_{td}(\mathcal{R})](c)), [id](c)\rangle\} \\
\longrightarrow_\rho & \{\langle [repeat*\,(Once_{td}(\mathcal{R}))](\emptyset), [id](b)\rangle, \langle [repeat*\,(Once_{td}(\mathcal{R}))](\emptyset), [id](c)\rangle\} \\
\longrightarrow_\rho & \{\langle \emptyset, \{b\}\rangle, \langle \emptyset, \{c\}\rangle\} \\
\longrightarrow_\rho & \{b, c\}
\end{array}
$$

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 unicity of the result. In our approach we do not stick to that situation 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 nor unicity of the normal form.

6 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 for the representation of non-conditional rewriting 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 previous section.

6.1 Definition of conditional rewriting

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

Definition 6.1 A *normal-boolean* 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 verifying $\mathcal{V}ar(r) \cup \mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$, 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 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-boolean 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], then the term u should be one of the two constants $True$ or $False$.

There are two possibilities to define the term c_ρ. If the reduction in a rewrite theory $\mathcal{R}$ is known we can define, as in Section 4.2, the term $[u_n](\ldots[u_1](c)\ldots)$ that evaluates to $\{c_\rho\}$. The second approach consists in defining the term c_ρ by using the normalization operators defined in Section 5. In this latter case we can define

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

Example 6.1 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 like shown in the 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

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

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. If the reduction of the condition c in the rewrite system contains the application of conditional rewrite rules then the corresponding ρ-term c_ρ contains the transformation of these conditional rules as described above.

Example 6.2 Let us assume that all the rules reducing inequalities between integers are defined for all pairs of integers and they are 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 $2 \geq 1$ reduces to $True$ by using the rewrite rule $(2 \geq 1) \to True$.

The corresponding reduction in the ρ-calculus is the following:

$[f(x) \to [True \to g(x)]([im(\mathcal{R}_<)](x \geq 1))](f(2)) \longrightarrow_{Fire}$
$\{[True \to g(2)]([im(\mathcal{R}_<)](2 \geq 1))\} \longrightarrow_{normalization}$
$\{[True \to g(2)](\{True\})\} \longrightarrow_{Batch}$
$\{\{[True \to g(2)](True)\}\} \longrightarrow_{Fire}$
$\{\{\{g(2)\}\}\} \longrightarrow_{Flat}$
$\{g(2)\}$

We give another example that handles conditional rewrite rules with the conditions normalized using conditional rewrite rules.

Example 6.3 We consider the set of rewrite rules $\mathcal{R}_f$ containing the rewrite rules $f(x) \to g(x)$ if $h(x) = b$ and $h(x) \to b$ if $x = a$. We denote by $\mathcal{R}_=$ the set containing the rewrite rule $(x = x) \to True$.

If we denote $\mathcal{R} = \mathcal{R}_f \cup \mathcal{R}_=$, the corresponding terms in ρ-calculus are

$$f(x) \to [True \to g(x)]([im(\mathcal{R})](h(x) = b))$$

and

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

The term $f(a)$ reduces to $g(a)$ using the rewrite rule $f(x) \to g(x)$ if $h(x) = b$, and we show below the corresponding reduction in ρ-calculus.

$$\begin{array}{ll} & [f(x) \to [True \to g(x)]([im(\mathcal{R})](h(x) = b))](f(a)) \\ \longrightarrow_\rho & \{[True \to g(a)]([im(\mathcal{R})](h(a) = b))\} \\ \longrightarrow_\rho & \{[True \to g(a)]([repeat*\,(Once_{bu}(\mathcal{R}))](h(a) = b))\} \\ \longrightarrow_\rho & \{[True \to g(a)](\{\langle [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](h(a) = b)), [id](h(a) = b)\rangle\})\} \end{array}$$

The term $[Once_{bu}(\mathcal{R})](h(a) = b))$ evaluates like follows

$$\begin{array}{ll} & [Once_{bu}(\mathcal{R})](h(a) = b)) \\ \longrightarrow_\rho & \{\langle\langle [Once_{bu}(\mathcal{R})](h(a)) = b, h(a) = [Once_{bu}(\mathcal{R})](b)\rangle, [\mathcal{R}](h(a) = b)\rangle\} \\ \longrightarrow_\rho & \{\langle\langle\{\langle\langle h([Once_{bu}(\mathcal{R})](a))\rangle, [\mathcal{R}](h(a))\rangle\} = b, h(a) = [Once_{bu}(\mathcal{R})](b)\rangle, [\mathcal{R}](h(a) = b)\rangle\} \\ \longrightarrow_\rho & \{\langle\langle\{\langle\langle h(\emptyset)\rangle, [\mathcal{R}](h(a))\rangle\} = b, h(a) = [Once_{bu}(\mathcal{R})](b)\rangle, [\mathcal{R}](h(a) = b)\rangle\} \\ \longrightarrow_\rho & \{\langle\langle\{[\mathcal{R}](h(a)) = b\}, h(a) = [Once_{bu}(\mathcal{R})](b)\rangle, [\mathcal{R}](h(a) = b)\rangle\} \end{array}$$

Now we evaluate $[\mathcal{R}](h(a))$ and we obtain

$$\begin{array}{ll} & [\mathcal{R}](h(a)) \\ \equiv & \{[\{h(x) \to [True \to b]([im(\mathcal{R})](x = a)), \\ & \quad f(x) \to [True \to g(x)]([im(\mathcal{R})](h(x) = b)), (x = x) \to True\}](h(a))\} \\ \longrightarrow_\rho & \{[h(x) \to [True \to b]([im(\mathcal{R})](x = a))](h(a)), \emptyset, \emptyset\} \\ \longrightarrow_\rho & \{[True \to b]([im(\mathcal{R})](a = a))\} \\ \longrightarrow_\rho & \{[True \to b]([repeat*\,(Once_{bu}(\mathcal{R}))](a = a))\} \\ \longrightarrow_\rho & \{[True \to b](\{\langle [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](a = a)), [id](a = a)\rangle\})\} \\ \longrightarrow_\rho & \{[True \to b](\{\langle [repeat*\,(Once_{bu}(\mathcal{R}))](\{True\}), [id](a = a)\rangle\})\} \\ \longrightarrow_\rho & \{[True \to b](\{\langle\{\langle [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](True)), [id](True)\rangle\}, [id](a = a)\rangle\})\} \\ \longrightarrow_\rho & \{[True \to b](\{\langle\{True\}, [id](a = a)\rangle\})\} \\ \longrightarrow_\rho & \{[True \to b](\{True\})\} \\ \longrightarrow_\rho & \{b\} \end{array}$$

We get back to the evaluation of $[Once_{bu}(\mathcal{R})](h(a) = b))$ and we get

$$\begin{array}{ll} & \{\langle\langle\{[\mathcal{R}](h(a)) = b\}, h(a) = [Once_{bu}(\mathcal{R})](b)\rangle, [\mathcal{R}](h(a) = b)\rangle\} \\ \longrightarrow_\rho & \{\langle\langle\{\{b\} = b\}, h(a) = [Once_{bu}(\mathcal{R})](b)\rangle, [\mathcal{R}](h(a) = b)\rangle\} \\ \longrightarrow_\rho & \{\langle\langle\{b = b\}, h(a) = [Once_{bu}(\mathcal{R})](b)\rangle, [\mathcal{R}](h(a) = b)\rangle\} \\ \longrightarrow_\rho & \{\langle\{\{b = b\}\}, [\mathcal{R}](h(a) = b)\rangle\} \\ \longrightarrow_\rho & \{b = b\} \end{array}$$

We continue with the evaluation of the initial term

$$\begin{array}{ll} & \{[True \to g(a)](\{\langle [repeat*\,(Once_{bu}(\mathcal{R}))]([Once_{bu}(\mathcal{R})](h(a) = b)), [id](h(a) = b)\rangle\})\} \\ \longrightarrow_\rho & \{[True \to g(a)](\{\langle [repeat*\,(Once_{bu}(\mathcal{R}))](\{b = b\}), [id](h(a) = b)\rangle\})\} \\ \longrightarrow_\rho & \{[True \to g(a)](\{\langle\{True\}, [id](h(a) = b)\rangle\})\} \\ \longrightarrow_\rho & \{[True \to g(a)](\{\{True\}\})\} \\ \longrightarrow_\rho & \{g(a)\} \end{array}$$

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

One should notice that $[True \to r](c)$ reduces to $\emptyset$ not only if c reduces to $False$ but also if c does not reduce to $True$, Therefore, we do not need a rewrite rule of the form $False \to x \neq y$ if $(x = y)$ in the set $\mathcal{R}_=$ used for normalizing the conditions.

We generalize the Lemma 4.2 and we state that an appropriate ρ-term represents any derivation in a conditional rewrite system:

Proposition 6.1 Given t and t' two terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ and $\mathcal{R}$ a conditional term rewrite system. If $t \xrightarrow{*}_{\mathcal{R}} t'$ then there exist ρ-terms $u_1, \ldots, u_n$ constructed starting from the rewrite rules in $\mathcal{R}$ such that $[u_n](\ldots[u_1](t)\ldots) \xrightarrow{*}_{\rho} \{t'\}$.

Proof: We consider rewrite rules of the form $l \to r$ if c applied to a term t at position p and thus, $t_{\lceil w \rceil_p} \longrightarrow_{\mathcal{R}} t_{\lceil \theta r \rceil_p} = t'$ if $\theta c \xrightarrow{*}_{\mathcal{R}} True$, where θ is the grafting such that $\theta l = w$. The corresponding ρ-term to be applied is $t_{\lceil l \to [True \to r](c_\rho) \rceil_p}$, with c_ρ constructed as explained above and such that when instantiated by θ it ρ-reduces to $\{True\}$. Since we need the reduction $\theta c \xrightarrow{*}_{\mathcal{R}} True$ to terminate in order to apply the conditional rewrite rule, the construction of c_ρ is possible.

The following derivation is obtained:

$$[t_{\lceil l \to [True \to r](c_\rho) \rceil_p}](t_{\lceil w \rceil_p}) \overset{Congruence^*}{\longrightarrow} \{t_{\lceil [l \to [True \to r](c_\rho)](w) \rceil_p}\} \overset{Fire}{\longrightarrow}$$

$$\{t_{\lceil \{\theta[True \to r](c_\rho)\} \rceil_p}\} = \{t_{\lceil \{[\theta True \to \theta r](\theta c_\rho)\} \rceil_p}\} = \{t_{\lceil \{[True \to \theta r](\theta c_\rho)\} \rceil_p}\} \overset{OpOnSet^*}{\longrightarrow}$$

$$\{\{t_{\lceil [True \to \theta r](\theta c_\rho) \rceil_p}\}\} \overset{Flat}{\longrightarrow} \{t_{\lceil [True \to \theta r](\theta c_\rho) \rceil_p}\} \overset{condition\ normalization}{\longrightarrow}$$

$$\{t_{\lceil [True \to \theta r](\{True\}) \rceil_p}\} \overset{Batch}{\longrightarrow} \{t_{\lceil \{[True \to \theta r](True)\} \rceil_p}\} \overset{OpOnSet}{\longrightarrow}$$

$$\{\{t_{\lceil [True \to \theta r](True) \rceil_p}\}\} \overset{Fire}{\longrightarrow} \{\{t_{\lceil \{\theta r\} \rceil_p}\}\} \overset{OpOnSet}{\longrightarrow} \{\{\{t_{\lceil \theta r \rceil_p}\}\}\} \overset{Flat}{\longrightarrow}$$

$$\{t_{\lceil \theta r \rceil_p}\}$$

Like in Lemma 4.2, if several rewrite rules are applied in order to get the term t', the same procedure is used for each of the rewrite rules, and the final strategy is the composition of these steps.

□

As for the non-conditional rewriting we can shortcut the procedure for representing the application of a conditional rewrite rule $l \to r$ if c on a term $t_{\lceil w \rceil_p}$ at position p and use the ρ-term

$$t_{\lceil [l \to [True \to r](c_\rho)](w) \rceil_p}$$

that reduces as above.

In the proof of Lemma 6.1 we have used the same approach as for describing the non-conditional rewriting. The operational approach consists in describing the corresponding reductions in the ρ-calculus using the operators defined in Section 5. According to the Corollary 5.1 we can state that if the term t normalizes to t' in the rewrite theory $\mathcal{R}$ then we can define the term $\varsigma(t) \equiv [im(\mathcal{R})](t)$ that normalizes in the ρ-calculus to the ρ-term $\{t'\}$.

Corollary 6.1 If the application of the rewrite rule $l \to r$ if c to the term u in the rewrite theory $\mathcal{R}$ evaluates to v then the ρ-term $[l \to [True \to r]([im(\mathcal{R})](c))](u)$ evaluates to $\{v\}$.

Starting from the results presented in Lemma 4.2 and in Lemma 6.1 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 rewrite 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, logic programming and automated theorem proving but also specification and exhaustive verification of authentication protocols [Pro].

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

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

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 [Mes92b, 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 $\texttt{dc}(s_1, \ldots, s_n)$) that returns the results of at most one non-deterministicly chosen unfailing strategy from its arguments and `dont know choose`(denoted $\texttt{dk}(s_1, \ldots, s_n)$) that returns all the possible results. A variant of the `dont care choose` operator is the `first choose` operator (denoted $\texttt{first}(s_1, \ldots, s_n)$) 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, $\texttt{repeat*}(s)$ 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 on the term a, ELAN provides two results: $a+1$ and $a+2$. When the strategy `first(x => x+1,x => x+2)` is applied on 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)` 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 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. The ELAN rewrite rules with local assignments can be given as well a ρ-calculus representation like in the following example:

Example 7.4 The ELAN's 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

$p_1 + p_2 \to [derive](p_1) + [derive](p_2)$,

$p_1 + p_2 \to [p_1' \to [p_2' \to p_1' + p_2'](p_2))]([derive](p_1))$.

The former representation syntacticly replaces all occurrences of local assigned variables in the right-hand side of the rule. In Example 7.5 we can notice that this representation yields the appropriate results even in a non-linear context where we can have sets with more than one element.

Since the representation of strategies is not in the scope of this paper we just mention some of the ρ-representations of the ELAN strategy operators. The concatenation ELAN operator ; and the iteration strategy operator `repeat*` are directly represented by the ρ-operators ";" and *repeat** respectively. The strategy `dk`$(s_1, \ldots, s_n)$ is represented in the ρ-calculus by the term $\{s_1, \ldots, s_n\}$. The ELAN strategy `first`$(s_1, \ldots, s_n)$ corresponds in the ρ-calculus to the ρ-term $first(s_1, \ldots, s_n)$.

Example 7.5 We consider the ELAN rule

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

Let us consider the strategy `derive` is `dk(a => b,a => c)`. Then, the application of the strategy *derive* on the term a gives the two results b and c. Thus, the application of the rule *deriveSum* on 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

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

that applied to a reduces as it follows

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

This set represents exactly the four results obtained in ELAN.

In the second representation from Example 7.4 we just bind the variables that are locally assigned in the rule by some variables that will be instantiated later with the results of the reduction of the local assignments.

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 $p'_2 \to p'_1 + p'_2$. The free variables in the right-hand side of a ρ-rewrite-rule allow us as well the parameterization of rewrite rules by strategies like in $y \to [f(x) \to [y](x)](f(a))$ where the strategy to be applied on x is not known in the rule $f(x) \to [y](x)$.

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 like in Example 7.6.

As for the conditional rewriting representation, we denote by c_ρ the ρ-term that, when instantiated by the proper substitution, corresponds to the normalization of the condition (term) c, instantiated accordingly, in the rewrite theory. In order to simplify the terms considered, in what follows, we denote by $\mathcal{R}_<$ the set of rewriting rules describing the inequalities.

Example 7.6 Let us consider the following ELAN rule:

```
[] x => y        where y := (dk(1 => 2,1 => 3)) x
                 if y >= 3                                  end
```

If we use an approach similar to the first one from Example 7.4 the variable y is replaced by its assignment in the right-hand side of the rewrite rule and in the condition. The above rule is represented by the ρ-term:

$$x \to [True \to [\{1 \to 2, 1 \to 3\}](x)]([\{1 \to 2, 1 \to 3\}](x) >= 3)$$

that, when applied to the term 1 yields the following derivation:

$$
\begin{array}{ll}
 & [x \to [True \to [\{1 \to 2, 1 \to 3\}](x)]([im(\mathcal{R}_<)]([\{1 \to 2, 1 \to 3\}](x) >= 3))](1) \\
\longrightarrow_{Fire} & \{[True \to [\{1 \to 2, 1 \to 3\}](1)]([im(\mathcal{R}_<)]([\{1 \to 2, 1 \to 3\}](1) >= 3))\} \\
\xrightarrow{*}_{Distrib} & \{[True \to \{[1 \to 2](1), [1 \to 3](1)\}]([im(\mathcal{R}_<)](\{[1 \to 2](1), [1 \to 3](1)\} >= 3))\} \\
\xrightarrow{*}_{Fire} & \{[True \to \{\{2\}, \{3\}\}]([im(\mathcal{R}_<)](\{\{2\}, \{3\}\} >= 3))\} \\
\xrightarrow{*}_{Flat} & \{[True \to \{2, 3\}]([im(\mathcal{R}_<)](\{2, 3\} >= 3))\} \\
\longrightarrow_{OpOnSet} & \{[True \to \{2, 3\}]([im(\mathcal{R}_<)](\{2 >= 3, 3 >= 3\}))\} \\
\longrightarrow_{cond.norm.} & \{[True \to \{2, 3\}](\{[im(\mathcal{R}_<)](2 >= 3), [im(\mathcal{R}_<)](3 >= 3)\})\} \\
\longrightarrow_{cond.norm.} & \{[True \to \{2, 3\}](\{False, True\})\} \\
\longrightarrow_{Batch} & \{\{[True \to \{2, 3\}](False), [True \to \{2, 3\}](True)\}\} \\
\longrightarrow_{Fire} & \{\emptyset, [True \to \{2, 3\}](True)\} \\
\longrightarrow_{Fire} & \{\emptyset, \{\{2, 3\}\}\} \\
\xrightarrow{*}_{Flat} & \{2, 3\}
\end{array}
$$

while, in ELAN, the correct result is: $\{3\}$.

The problem in the Example 7.6 is the double evaluation of the local variable y: once in the condition and once in the right-hand side of the rule. If the local variable is evaluated to a set of results this set will be returned if one of its elements satisfies the condition, while the appropriate result would be the subset of

elements satisfying the condition. Therefore, we need a mechanism that evaluates only once each of the local assignments of a rule.

We denote by $t_{\lceil s_1 \rceil_{p_1} \cdots \lceil s_n \rceil_{p_n}}$ the term t containing the sub-terms $s_1, \ldots, s_n$ at the positions $p_1, \ldots, p_n$ respectively. Without loosing generality, we consider that an ELAN rule that has the following form:

$$\begin{array}{lll} \text{[label]} & l & \Rightarrow \quad r_{\lceil x_1 \rceil_{q_1} \lceil x_2 \rceil_{q_2}} \\ & & \quad \text{where } x_1 := (s_1)t_1 \\ & & \quad \text{if } C_{1\lceil x_1 \rceil_{p_1}} \\ & & \quad \text{where } x_2 := (s_2)t_2 \\ & & \quad \text{if } C_{2\lceil x_1 \rceil_{r_1} \lceil x_2 \rceil_{r_2}} \qquad \text{end} \end{array}$$

where t_{i+1} can depend on x_i.

If we use an approach similar to the ρ-representation of conditional rules, the ELAN rule presented above is expressed as the ρ-term:

$$\begin{array}{l} l \to [x_1 \to [\{True \to [x_2 \to \quad [\{True \to r_{\lceil x_1 \rceil_{q_1} \lceil x_2 \rceil_{q_2}}, \\ \qquad\qquad\qquad\qquad\qquad\qquad False \to \emptyset\}]([im(\mathcal{R})](C_{2\lceil x_1 \rceil_{p_1} \lceil x_2 \rceil_{p_2}})) \\ \qquad\qquad\qquad\qquad\quad]([s_2](t_2)), \\ \qquad\qquad\quad False \to \emptyset\}]([im(\mathcal{R})](C_{1\lceil x_1 \rceil_{p_1}})) \\ \qquad]([s_1](t_1)) \end{array}$$

or the simpler one:

$$\begin{array}{l} l \to [x_1 \to [True \to [x_2 \to \quad [True \to r_{\lceil x_1 \rceil_{q_1} \lceil x_2 \rceil_{q_2}}] \\ \qquad\qquad\qquad\qquad\qquad\qquad ([im(\mathcal{R})](C_{2\lceil x_1 \rceil_{p_1} \lceil x_2 \rceil_{p_2}})) \\ \qquad\qquad\qquad\qquad]([s_2](t_2))] \\ \qquad\qquad ([im(\mathcal{R})](C_{1\lceil x_1 \rceil_{p_1}})) \\ \qquad]([s_1](t_1)) \end{array}$$

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

We notice that, when evaluating the application of such a rule, the evaluation rule $Fire$ of the ρ-calculus should be used for an application of the form $[True \to u](v)$ only when the term v is completely instantiated and reduced. Since the term $True$ has only functional position at the top, the previous condition corresponds exactly to the strategy defined in Section 3.3 in order to obtain a confluent ρ-calculus.

The way this transformation applies on an ELAN rewrite rule and the corresponding evaluation of the obtained ρ-term are illustrated in Example 7.7.

Example 7.7 For the following ELAN rewrite rule

```
x  =>  h(z,l(y))
       where y:= (s1) x
       where z:= (s2) x
       if y > z            end
```

with the strategies `s1` and `s2` equal to `dk(2 => 3, 2 => 4)`, the corresponding ρ-term is:

$$x \to [y \to [z \to [True \to h(z, l(y))]([im(\mathcal{R}_<)](y > z))]([s_2](x))]([s_1](x))$$

If we represent by the set $\{2 \to 3, 2 \to 4\}$ the strategy `dk(2 => 3, 2 => 4)` the application of the rule on the term 2 yields the following derivation:

$$\begin{array}{ll} & [x \to [y \to [z \to [True \to h(z, l(y))]([im(\mathcal{R}_<)](y > z))]([s_2](x))]([s_1](x))](2) \\ \longrightarrow_{Fire} & \{[y \to [z \to [True \to h(z, l(y))]([im(\mathcal{R}_<)](y > z))]([s_2](2))]([\{2 \to 3, 2 \to 4\}](2))\} \\ \longrightarrow_{Distrib+Fire} & \{[y \to [z \to [True \to h(z, l(y))]([im(\mathcal{R}_<)](y > z))]([s_2](2))](\{3, 4\})\} \\ \longrightarrow_{Batch} & \{[y \to [z \to [True \to h(z, l(y))]([im(\mathcal{R}_<)](y > z))]([s_2](2))](3), \\ & [y \to [z \to [True \to h(z, l(y))]([im(\mathcal{R}_<)](y > z))]([s_2](2))](4)\} \\ \longrightarrow_{Fire} & \{\{[z \to [True \to h(z, l(3))]([im(\mathcal{R}_<)](3 > z))]([\{2 \to 3, 2 \to 4\}](2))\}, \\ & \{[z \to [True \to h(z, l(4))]([im(\mathcal{R}_<)](4 > z))]([\{2 \to 3, 2 \to 4\}](2))\}\} \\ \longrightarrow_{Distrib+Fire} & \{\{[z \to [True \to h(z, l(3))]([im(\mathcal{R}_<)](3 > z))](\{3, 4\})\}, \\ & \{[z \to [True \to h(z, l(4))]([im(\mathcal{R}_<)](4 > z))](\{3, 4\})\}\} \\ \longrightarrow_{Batch} & \{[z \to [True \to h(z, l(3))]([im(\mathcal{R}_<)](3 > z))](3), \\ & [z \to [True \to h(z, l(3))]([im(\mathcal{R}_<)](3 > z))](4), \\ & [z \to [True \to h(z, l(4))]([im(\mathcal{R}_<)](4 > z))](3), \end{array}$$

$$
\begin{array}{ll}
 & [z \to [True \to h(z, l(4))]([im(\mathcal{R}_<)](4 > z))](4)\} \\
\longrightarrow_{Fire} & \{[True \to h(3, l(3))]([im(\mathcal{R}_<)](3 > 3)),\ [True \to h(4, l(3))]([im(\mathcal{R}_<)](3 > 4)), \\
 & [True \to h(3, l(4))]([im(\mathcal{R}_<)](4 > 3)),\ [True \to h(4, l(4))]([im(\mathcal{R}_<)](4 > 4))\} \\
\longrightarrow_{cond.red.} & \{[True \to h(3, l(3))](\{False\}),\ [True \to h(4, l(3))](\{False\}), \\
 & [True \to h(3, l(4))](\{True\}),\ [True \to h(4, l(4))](\{False\})\} \\
\longrightarrow_{Fire} & \{\emptyset, \emptyset, \{h(3, l(4))\}, \emptyset\} \\
\longrightarrow_{Flat} & \{h(3, l(4))\}
\end{array}
$$

This representation not only allows a correct transformation of ELAN reduction 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 like 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 until now. 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 on its normal form. Moreover, each time a labeled rewrite rule is applied on 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

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

should be represented in the ρ-calculus by the term

$$p_1 \to [im(\mathcal{R})]([p_1' \to p_1'][derive]([im(\mathcal{R})](p_1))))$$

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

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 used in Example 7.3 can be immediately represented by the ρ-term

$$attStrat \to repeat*(\{initiate, \ldots, intruder\}); attackFound$$

where we assume that the rewrite rules have been already given a ρ-representation.

In Example 7.9 we present an ELAN module and the ρ-interpretations of all the rewrite rules and strategies involved.

Example 7.9 We take the following ELAN module

```
module named
import global cmp[elem];end
sort elem ;end

operators global
  f(@) : (elem)  elem;
  @ # @ : (elem elem)  elem;
  a : elem; b : elem; c : elem; d : elem;
  e : elem; m : elem; n : elem; p : elem;
end

rules for elem
    x,y : elem;
global
    [r1] f(x) =>  y # a
```

```
                  where y := (s1) x # b
                  if x # y == d                end
    [r2] d => e                                end
    [r3] c => e                                end
    [r4] e => m                                end
end

rules for elem
x,y : elem;
global
    [] m => n                              end    //ur1
    [] n # a => p                          end    //ur2
    [] c # n => d                          end    //ur3
end

strategies for elem
implicit
    []strat =>  r1                         end      //sr1
    []s1 =>  s2;s3                         end      //sr2
    []s2 =>  dk(r2,r3)                     end      //sr3
    []s3 =>  r4                            end      //sr4
end

end
```

We denote by $\mathcal{C}$ the set of unlabeled rewrite rules from the module `cmp[elem]`. The unlabeled rewrite rules are clearly described by the following ρ-terms $m \to n$, $n\#a \to p$, $c\#n \to d$ and they will be referred by the labels $ur1, ur2, ur3$. The unlabeled rewrite rules defining the strategies are represented by the ρ-terms $strat \to r1$, $s1 \to s2; s3$, $s2 \to dk(r2, r3)$, $s3 \to r4$ and they will be referred by the labels $sr1, sr2, sr3, sr4$.

The rule $r1$ is represented by the ρ-term

$$\begin{array}{l} f(x) \to [im(\{ur1, ur2, ur3\})](\\ \quad [y \to [True \to y\#a](\\ \qquad\qquad [im(\{ur1, ur2, ur3\} \cup \mathcal{C})](x\#y == d)) \\ \quad]([[im(\{sr1, sr2, sr3, sr4\})](s1)]([im(\{ur1, ur2, ur3\})](x\#b))) \end{array}$$

The last line from the above representation represents the assignment to the variable y of the term $(x\#b)$ on which we apply the strategy $s1$ normalized according to the set of rules for strategies $\{sr1, sr2, sr3, sr4\}$. The result of the application of the strategy on the term is normalized according to the set of unlabeled rewrite rules $\{ur1, ur2, ur3\}$. On the third line the condition is normalized according to the unlabeled rewrite rules from the current module and from the appropriate imported modules. In our case the module `cmp[elem]` contains the rewrite rules dealing with the equality and thus the set of ρ-rewrite rules $\mathcal{C}$ is used. If the condition is satisfied the variable y from the term $y\#a$ is instantiated accordingly and the term obtained is normalized with respect to the set of rewrite rules $\{ur1, ur2, ur3\}$.

In the same line the rules $r2, r3, r4$ are represented by the ρ-terms

$$\begin{array}{lcl} d & \to & [im(\{ur1, ur2, ur3\})](e), \\ c & \to & [im(\{ur1, ur2, ur3\})](e), \\ e & \to & [im(\{ur1, ur2, ur3\})](m). \end{array}$$

The strategy rules are represented in a similar way by the ρ-terms

$$\begin{array}{lcl} strat & \to & r1, \\ s1 & \to & [im(\{sr1, sr2, sr3, sr4\})](s2); [im(\{sr1, sr2, sr3, sr4\})](s3), \\ s2 & \to & \{r2, r3\}, \\ s3 & \to & r4. \end{array}$$

The ELAN application of the strategy $strat$ on the term $f(c)$ gives as result the term p and the corresponding ρ-term

$$[[im(\{sr1, sr2, sr3, sr4\})](strat)](f(c))$$

yields the result $\{p\}$.

We have seen in Example 7.9 how the evaluation mechanism of ELAN can be explicitly described using the appropriate ρ-terms. This example presents only part of the ELAN operators; a complete description of the language using the ρ-calculus will be given in a forthcoming paper.

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 ρ_T-calculus, where the evaluation rules are not guided by a strategy, is not confluent. When an appropriate but rather natural generalized (i.e. lazy) call by value evaluation strategy is used the confluence is recovered.

The ρ_T-calculus is both conceptually simple as well as very expressive. This allowed us to represent the terms and reductions from λ-calculus and conditional rewriting. Using appropriate ρ-definitions for term traversal operators and a fixed point operator we are able 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 give a semantics to ELAN rules. 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.

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, we have shown that the ρ_T-calculus allows to see the rewriting arrow $\rightarrow$ as a generalization of the *lambda* notation, making the latter unnecessary when combining lambda-calculus with rewriting. To go further in the study and the use of the ρ_T-calculus for the combination of first-order and higher-order paradigms, we plan to investigate the relationship of this calculus with higher-order rewrite concepts like CRS and HOR [OR93].

Another main issue is of course to understand how a typed version of the ρ_T-calculus calculus could be defined in order to obtain all the good standard properties like subject reduction and strong normalization.

Among the many other topics of further research, let us finally mention the deepening of the relationship between the ρ_T-calculus and the rewriting logic [Mes92b], the study of the models of the ρ_T-calculus, and also a better understanding of the relationship between the rewriting relation and the rewriting calculus.

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 and Vincent van Oostrom for suggestions and pointers to the literature.

References

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

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[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.

[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 sf ELAN. In *Workshop on Modelling and Verification*, Besancon, France, December 1999.

[CK99a] 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.

[CK99b] H. Cirstea and C. Kirchner. The confluence of the rewriting calculus. Rapport de recherche, Institut National de Recherche en Informatique et en Automatique, July 1999. Forthcoming.

[CK99c] H. Cirstea and C. Kirchner. An explicit version of the rewriting calculus. Rapport de recherche, Institut National de Recherche en Informatique et en Automatique, July 1999. Forthcoming.

[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.

[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.

[Eke93] S. Eker. Improving the efficiency of AC matching and unification. Research report 2104, INRIA, Inria Lorraine & Crin, November 1993.

[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)*, Lecture Notes in Computer Science. Springer-Verlag, 1983.

[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.

[KK98] C. Kirchner and H. Kirchner. Rewriting, solving, proving. Notes de cours de l'école jeunes chercheurs en programmation, 1998.

[KKR90a] C. Kirchner, H. Kirchner, and M. Rusinowitch. Constrained first order logic. Technical report, Centre de Recherche en Informatique de Nancy, March 1990.

[KKR90b] 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.

[Mes89] J. Meseguer. General logics. In *Proc. Logic Colloquium '87*. North Holland, 1989.

[Mes92a] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes92b] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 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.

[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.

[OR93] V. v. Oostrom and F. f. 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.

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

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

[Pro] T. Protheo. The inELAN home page. WWW page, LORIA. `http://www.loria.fr/ELAN`.

[Rin94] C. Ringeissen. Combination of matching algorithms. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen (France)*, volume 775 of *Lecture Notes in Computer Science*, pages 187–198. Springer-Verlag, February 1994.

[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.

[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.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399