Electronic Notes in Theoretical Computer Science 86 No. 4 (2004)
URL: http://www.elsevier.nl/locate/entcs/volume86.html 33 pages

Rewrite Strategies in the Rewriting Calculus

Horatiu Cirstea
LORIA & University Nancy II[1]

Claude Kirchner
LORIA & INRIA[1]

Luigi Liquori
INRIA Sophia-Antipolis[2]

Benjamin Wack
LORIA & University Henri Poincaré[1]

Abstract

This paper presents an overview on the use of the rewriting calculus to express rewrite strategies. We motivate first the use of rewrite strategies by examples in the ELAN language. We then show how this has been modeled in the initial version of the rewriting calculus and how the matching power of this framework facilitates the representation of powerful strategies.

1 Introduction

The notion of strategy appears in all human activities under many different names: strategies, tactics, plans, road-maps, ... to mention just a few. This always means that one wants to describe in a concise manner how to reach a given goal in an environment where many different possibilities can arise and where consequently one needs to search rather than just compute.

In computer science the strategy concept spread-off everywhere, from artificial intelligence to logic or semantics. When modeling a certain situation like planning for the moves of a robot on Mars, the inferences to be made by a theorem prover or the evaluation rules of a programming language, the notion of rewrite rule naturally arises.

Indeed, a so called rewrite rule consists of a pattern that describes a schematic situation and the transformation that should be applied in the given

[1] Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex France
[2] 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis France

case. Again, a natural model for defining the pattern is to use term objects but other objects like matrices or graphs could also be considered. So, in this paper, we are considering rewrite rules consisting of a pair of terms that model the class of objects to be transformed and the objects in which they should be actually transformed.

What makes the rewrite rule concept so attractive is twofold. First it is schematic: one needs only to describe a schema and not all its instances. Consequently, to decide if a rewrite rule applies requires the use of a procedure to check that the schema is applicable, *i.e.* a matching algorithm. Second it is local: the application of a rewrite rule does not depend on the context and thus can be decided locally.

Therefore, rewrite rules are very convenient for describing schematically and locally the transformations one wants to operate. This is used in many situations like dealing with equational logic, performing inferences in theorem proving, or when performing computations. But in addition to the locality of rule application, one main ingredient should be added to make the concept operational and allow one to answer the questions: which rule should be applied and where? This fundamental ingredient is a *rewrite strategy*.

As such, the notion of rewrite strategy is in everyday use: from normalization strategies to optimal strategies, from leftmost-innermost to lazy ones. Typically, programming languages use call by value or lazy strategies. Automated theorem provers use depth-first or breadth-first proof search strategies. In many cases these strategies are built-in and the users of the programming language or of the prover have no way to adapt them to their specific use.

Since strategies allow us to control the schematically and locally described transformations, it is conceptually and practically fundamental to permit the users to define their own ones. This is what proof assistants like LCF, Coq, ELF permit under the name of tactics and tacticals. In programming languages such a control can be reached using reflexivity like in LISP and Maude or explicitly using a dedicated language like in ELAN or Stratego.

In this last kind of languages, user-defined strategies are defined and executed using the strategy combinators provided by the language. *How to design such a language and its semantics is the topic of this paper.*

Historically, we started in designing the ELAN language whose aim was to easily prototype the combination of deduction and computation mechanisms as needed in constraint solving and automated theorem proving. This leaded us to a first semantics of the language using rewriting logic [22,4] and then to the design of a very general calculus able to take into account all the aspects of the language; we call it the rewriting calculus [8,6]. As we shortly recall later in this paper, the rewriting calculus generalizes the lambda-calculus as abstractions can be made not only on variables but also on terms.

An important feature of the rewriting calculus is its ability to model rewriting strategies. What was needed for this purpose was: (i) to have rewrite rules

as primal strategies, (ii) to return explicitly sets of terms to model the fact that equational rewriting can produce sets of results. The explicit handling of result sets also allowed us to detect that a rewrite step can not be applied at some occurrence, since in this case the result set is just empty. Then we had to iterate rewritings in a term and therefore, we naturally added a (iii) fix-point iterator, and since the rewriting calculus embeds the lambda one, we first used standard fix-points like Turing's one. One important feature that escaped from the scope of this initial design was the capability to stop a computation as soon as a first rewrite is successfully applied. We therefore (iv) enriched the rewriting calculus with a strategy called *first*, which returns the results of the first non-failing computation. In this initial setting we were able to describe useful strategies like the innermost or outermost ones. The fact that a specific failure operator makes possible the definition of the *first* operator was showed later and all this is described at the end of the third section of this paper.

From this initial design that we denote in this paper Rho Calculus, by simplifying the syntax and the operational semantics, we got a simpler expression of the rewriting calculus that we denote in this paper ρCal. At that point, we also better exploited the expressive power of the calculus (*e.g.* for encoding object calculi) and introduced sophisticated type systems. This leaded us to the discovery of a quite powerful feature of the simply typed version of the rewrite calculus: because of its matching power, the calculus allows indeed some well-typed terms to be non-normalizing. One nice application of this is the definition of well-typed and very concise fix-points based on the matching capability of the calculus. As a consequence we can describe strategies in a very powerful language based on these matching based fix-points. This is described in the last part of this paper together with applications to the description of rewrite strategies and to the combination of such strategies.

The new simplified syntax, together with a deterministic call-by-value operational semantics for the calculus, allows us to detect matching failures more easily. The encoding of the *first* by clever use of an exception catching mechanism is then available. Various options are available for exception handling, depending on the new constructs we introduce and on the semantics we put on them. The specific mechanisms related to exceptions and the behavior of the latest *first* expression are detailed at the end of Section 4.

2 ELAN: A Language which Deals with Strategies

The ELAN system [25,20,5] provides an environment for specifying and prototyping deduction systems in a language based on rules controlled by strategies. Its purpose is to support the design of theorem provers, logic programming languages, constraint solvers and decision procedures and to offer a modular framework for studying their combination.

ELAN takes from functional programming the concept of abstract data

type and the function evaluation principle based on rewriting. But rewriting is inherently non-deterministic since several rules can be applied at different positions in a same term, and in ELAN, a computation may have several results. This aspect is taken into account through choice operators and a backtracking capability. One of the main originalities of the language is to provide strategy constructors that specify whether an evaluation returns several, at least one or only one result. This declarative handling of non-determinism is part of a strategy language allowing the programmer to specify the control on rule application. This is in contrast to many existing rewriting-based languages where the term reduction strategy is hard-wired and not accessible to the designer of an application. The strategy language offers primitives for sequential composition, iteration, deterministic and non-deterministic choices of elementary strategies that are labeled rules. From these primitives, more complex strategies can be expressed. Additionally, the user can introduce new strategy operators and define them by rewrite rules.

2.1 The Specification Language

The specification formalism provided in the ELAN system is close to the algebraic specification formalism. Signatures introduce sorts of data and corresponding operators. Operators can be defined using a mixfix syntax and can be declared as associative and commutative; the associativity and commutativity axioms are called structural axioms and their application is embedded into the matching process.

```
module boolean
sort Bool; end
operators global
  true      :             Bool;
  false     :             Bool;
  @ and @   : (Bool Bool) Bool  (AC);
  @ or  @   : (Bool Bool) Bool  (AC);
    not @   : (Bool )     Bool;
end
```

In the algebraic style, the semantics is described by a set of first-order formulas. In ELAN, the formulas are a very general form of rewrite rules with conditions and local evaluations. For instance, simple rewrite rules for booleans are given as follows:

```
rules for Bool
  P : Bool;
global
  [] true  or P  => true      end
  [] false or P  => P         end
  [] true  and P => P         end
  [] false and P => false     end
  [] not true    => false     end
```

```
  [] not false   => true      end
end
```

The two values `true` and `false` are said irreducible or in normal form. This set of rules is terminating and confluent, which ensures that any boolean formula has a unique normal form. For such systems, it is not needed to specify in which order the rules are applied, nor at which position in the term. The ELAN system adopts in such a case a strategy by default which selects the leftmost and innermost redex at each step. However in many situations, and especially to deal with non-confluent or non-terminating rewrite systems, it is suitable to express which rule to apply.

For specifying this kind of control, ELAN introduces the possibility to name rules, using brackets in front of a rule to enclose its name. In the previous boolean example, the names are unspecified and such rules are said unlabeled. The labeled rules and strategies are applied at the top of a term. In order to apply strategies on subterms, the syntax of rewrite rules has been enriched by local evaluations, used to call strategies and to specify conditions of application. We concentrate in this paper on the application of (basic) rules and strategies and therefore we do not present here the syntax of the local evaluation constructions.

The capability of specifying control is a main originality of ELAN compared to other specification languages. Let us explain in more details how to build strategies that compute one or several results, specify the order of applied rules, or iterate as much as possible the application of a strategy or a rule on a term.

2.2 Strategy Specification

A labeled rule is the most elementary strategy and is called a primal strategy. The result of applying a rule labeled lab on a term t is a set of terms. Note that there may be several rules with the same label. If no rule labeled lab applies on the term t, the set of results is empty and we say that the rule lab fails. To understand why applying one rule at the top of a term can yield several results, one has to know that local assignments in a rewrite rule can call strategies on subterms. If the strategy in a local assignment has several results, so has the rewrite rule. A labeled rule lab can be considered as the simplest form of a strategy which returns all results of the rule application. As any strategy, lab can also be encapsulated by an operator dc one that returns a non-deterministically chosen result. In that case, dc one(lab) returns at most one result. In addition ELAN provides a few built-in strategy operators that take one or several strategies as arguments and can be used to build new strategies:

- 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;

- the dk operator, with a variable arity, is an abbreviation of *dont know choose*. dk$(S_1, \ldots, S_n)$ takes all strategies given as arguments, and returns, for each of them the set of all its results. dk$(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail;
- the dc operator, with a variable arity, is an abbreviation of *dont care choose*. dc$(S_1, \ldots, S_n)$ selects only one strategy that does not fail among its arguments, say S_i, and returns all its results. dc$(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail. How to choose S_i is not specified;
- a specific way to choose an S_i is provided by the first operator that selects the first strategy that does not fail among its arguments, and returns all its results. So if S_i is selected, this means that all strategies $S_1, \ldots, S_{i-1}$ have failed. Again first$(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail;
- if only one result is wanted, one can use the operators first one or dc one that select a non-failing strategy among their arguments (either the first or anyone respectively), and return a non-deterministically chosen result of the selected strategy;
- id is the identity strategy that does nothing, and never fails;
- fail always fails and returns an empty set of results;
- repeat*(S) iterates the strategy S until it fails and then returns the last obtained result. repeat*(S) never fails and terminates only when S fails;
- iterate*(S) is similar to repeat*(S), except that it returns all intermediate results of successive applications of S.

In addition to these primitive strategy operators, the user can define new strategy operators and strategy rules for their evaluation [3].

Example 2.1 If the strategy `dk(x=>x+1,x=>x+2)` is applied to the term a, ELAN provides two results: $a + 1$ and $a + 2$. When `first(x=>x+1,x=>x+2)` is applied to the same term only the $a + 1$ result is obtained. The strategy `first(b=>b+1,a=>a+2)` applied to the term a yields the result $a + 2$.

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

3 The Initial Design of the Rewriting Calculus

We first gave a semantics to ELAN and in particular to its strategy language by using a rewriting logic [4]. But some fine aspects of the strategy language escaped from a simple use of the rewriting logic features. This and general considerations on the integration of lambda-calculus together with first and higher-order rewriting leaded to the design of the rewriting calculus as introduced in [7,8]. We briefly present here the initial design of the Rho Calculus and we define several operators allowing us to define different classical rewriting strategies like, for example, innermost or outermost.

3.1 *First Version of the* Rho Calculus

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 $\mathcal{F}_m$ 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 is inductively defined by the following grammar:

$$\mathcal{T} ::= \mathcal{X} \mid f(\mathcal{T}, \ldots, \mathcal{T}) \mid \{\mathcal{T}, \ldots, \mathcal{T}\} \mid [\mathcal{T}](\mathcal{T}) \mid \mathcal{T} \twoheadrightarrow \mathcal{T}$$

In what follows we suppose that $s, t, u, v, \ldots \in \mathcal{T}$, and $x, y, \ldots \in \mathcal{X}$, and $f, g, \ldots \in \mathcal{F}$. These symbols can be also indexed.

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

The main intuition behind this syntax is that a rewrite rule is an abstraction, the left-hand side of which determines the bound variables and some contextual information. Having new variables in the right-hand side is just the ability to have free variables in the calculus. One can notice that the λ-terms [2] and standard first-order rewrite rules [12,1] are clearly objects of this calculus. For example, the λ-term $\lambda x.(y\ x)$ corresponds to the ρ-term $x \twoheadrightarrow [y](x)$ 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 representing the potential non-determinism. A set of terms can be seen as the set of distinct results obtained by applying a rewrite rule to a term. Other choices could be made depending on the intended use of the calculus. For example, if we want to provide all the results of an application, including the identical ones, a multi-set could be used. When the order of the computation of the results is important, lists could be employed. Since in this presentation of the calculus we focus on the possible results of a computation and not on their number or order, sets are used.

Example 3.1 If we consider $\mathcal{F}_0$=$\{a, b, c\}$, $\mathcal{F}_1$=$\{f\}$, $\mathcal{F}_2$=$\{g\}$, $\mathcal{F}$=$\mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2$ and x, y variables in $\mathcal{X}$, some ρ-terms are:

- $[g(x, y) \twoheadrightarrow f(x)](g(a, b))$; a classical rewrite rule application.
- $[y \twoheadrightarrow [x \twoheadrightarrow x + y](b)]([x \twoheadrightarrow x](a))$; a ρ-term that corresponds to the λ-term $(\lambda y.((\lambda x.x + y)\ b))\ ((\lambda x.x)\ a)$. In the rewrite rule $x \twoheadrightarrow x + y$ the variable y is free but in the rewrite rule $y \twoheadrightarrow [x \twoheadrightarrow x + y](b)$ this variable is bound.

- $[x \rightarrowtriangle [x](x)](x \rightarrowtriangle [x](x))$; the well-known Ω λ-term. As in the λ-calculus, we will see that the evaluation of this term is not terminating in the Rho Calculus.

Computing the matching substitutions from a ρ-term t to a ρ-term u is an important parameter of the Rho Calculus.

For a given theory $\mathbb{T}$ over ρ-terms, a $\mathbb{T}$-*match-equation* is a formula of the form $t \lessdot_{\mathbb{T}} u$, where t and u are ρ-terms. A substitution σ is a solution of the $\mathbb{T}$-match-equation $t \lessdot_{\mathbb{T}} u$ if $\mathbb{T} \models \sigma(t) \equiv u$. A $\mathbb{T}$-*matching system* is a conjunction of $\mathbb{T}$-match-equations. A substitution is a solution of a $\mathbb{T}$-matching system P if it is a solution of all the $\mathbb{T}$-match-equations in P. We denote by $\mathbb{F}$ a $\mathbb{T}$-matching system without solution. A $\mathbb{T}$-matching system is called *trivial* when all substitutions are solution of it. We define the function $\mathcal{S}ol$ on a $\mathbb{T}$-matching system $\mathcal{S}$ as returning the set of all $\mathbb{T}$-matches of $\mathcal{S}$ when $\mathcal{S}$ is not trivial and $\{\sigma_{\mathsf{id}}\}$, where σ_{id} is the identity substitution, when $\mathcal{S}$ is trivial. Notice that when the matching system has no solution the function $\mathcal{S}ol$ returns the empty set. By abuse of notation we denote by $\sigma_{(t \lessdot u)}$ the unique solution of the matching equation $t \lessdot u$ if $\mathcal{S}ol(t \lessdot u)$ is a singleton.

Since in general we could consider arbitrary theories over ρ-terms, $\mathbb{T}$-matching is in general undecidable, even when restricted to first-order equational theories [17]. But we are interested here in the decidable cases. For example when $\mathbb{T}$ is empty, the syntactic matching substitution from t to u, when it exists, is unique and can be computed by a simple recursive algorithm given for example by G. Huet [16,19].

Since we are dealing with "$\rightarrowtriangle$" as a binder, like for any calculus involving binders (as the λ-calculus), α-conversion should be used to obtain a correct substitution calculus and the first-order substitution (called here *grafting*) is not directly suitable for the Rho Calculus. We consider the usual notions of α-conversion and higher-order substitution as defined, for example, in [13].

The set of evaluation rules of the Rho Calculus is recalled in Figure 1, where we assume we are given a theory $\mathbb{T}$ over ρ-terms having a decidable matching problem.

As defined by the evaluation rule $(Fire)$, 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. When the matching yields a failure represented by an empty set of substitutions, the result of the application of the rule $(Fire)$ is the empty set. This rule, like all the evaluation rules of the calculus, can be applied at any position of a ρ-term.

We should point out that, as in λ-calculus, an application can always be evaluated. But, unlike in λ-calculus, the set of results can be empty. More generally, when matching modulo a theory $\mathbb{T}$, the set of resulting matches may be empty, a singleton (as in the empty theory), a finite set (as for associativity-commutativity) or infinite (see [14]). We have thus chosen to represent the result of a rewrite rule application to a term as a set. An empty set means

$$
\begin{array}{lrcl}
(Fire) & [l \twoheadrightarrow r](t) & \to & \{\sigma_1 r, \ldots, \sigma_n r, \ldots\} \\
 & & & \text{where } \{\sigma_1, \ldots, \sigma_n, \ldots\} = \mathcal{S}ol(l \ll_{\mathbb{T}} t) \\
(Cong) & [f(u_1, \ldots, u_n)](f(v_1, \ldots, v_n)) & \to & \{f([u_1](v_1), \ldots, [u_n](v_n))\} \\
(Cong\_fail) & [f(u_1, \ldots, u_n)](g(v_1, \ldots, v_m)) & \to & \emptyset \\
(Distrib) & [\{u_1, \ldots, u_n\}](v) & \to & \{[u_1](v), \ldots, [u_n](v)\} \\
(Batch) & [v](\{u_1, \ldots, u_n\}) & \to & \{[v](u_1), \ldots, [v](u_n)\} \\
(Switch_L) & \{u_1, \ldots, u_n\} \twoheadrightarrow v & \to & \{u_1 \twoheadrightarrow v, \ldots, u_n \twoheadrightarrow v\} \\
(Switch_R) & u \twoheadrightarrow \{v_1, \ldots, v_n\} & \to & \{u \twoheadrightarrow v_1, \ldots, u \twoheadrightarrow v_n\} \\
(OpOnSet) & f(v_1, \ldots, \{u_1, \ldots, u_m\}, \ldots, v_n) & \to & \\
 & & & \{f(v_1, \ldots, u_1, \ldots, v_n), \ldots, f(v_1, \ldots, u_m, \ldots, v_n)\} \\
(Flat) & \{u_1, \ldots, \{v_1, \ldots, v_n\}, \ldots, u_m\} & \to & \{u_1, \ldots, v_1, \ldots, v_n, \ldots, u_m\}
\end{array}
$$

Fig. 1. The evaluation rules of the Rho Calculus

that the rewrite rule $l \twoheadrightarrow r$ fails to apply to t in the sense of a matching failure between l and t.

In order to push rewrite rule application deeper into terms, we introduce the two $(Congruence)$ evaluation rules. 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.

The rules $(Distrib)$ and $(Batch)$ describe the interaction between the application and the set operators, the rules $(Switch_L)$ and $(Switch_R)$ describe the interaction between the abstraction and the set operators, the rule $(OpOnSet)$ describe the interaction between the symbols of the signature and the set operators.

We usually care about the set of results obtained by reducing the redexes and not about the exact trace of the reduction leading to these results. We use the evaluation rule $(Flat)$ that flattens the sets and eliminates the (nested) set symbols. Notice that this implies that failure (the empty set) is *not* strictly propagated on sets.

The strategy guiding the application of the evaluation rules is crucial for obtaining good properties for the Rho Calculus such as confluence. It has been shown [8] that if the rule $(Fire)$ is applied under no conditions at any position of a ρ-term, confluence does not hold.

The main reason for confluence failure comes from 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. The reasons for the non-confluence of the calculus are explained in [8] and a solution is proposed for obtaining a confluent calculus. The confluent strategy can be given explicitly or as a condition on the application of the rule $(Fire)$.

Since the sets (empty or having more than one element) are the main cause of non-confluence of the calculus, a natural strategy consists in reducing the application of a rewrite rule by respecting the following steps: instantiate and reduce the argument of the application, push out the resulting set braces by distributing them in the terms and only when none of the previous reductions is possible, use the evaluation rule $(Fire)$. We can easily express this strategy by imposing a simple condition for the application of the evaluation rule $(Fire)$: the evaluation rule $(Fire)$ is applied to a redex $[l \twoheadrightarrow r](t)$ only if the terms l, t are first order ground terms. Less restrictive strategies guaranteeing the confluence are defined in [8].

3.2 Defining Strategy Operators in Rho Calculus

It is shown in [8] that for any reduction in a rewrite theory there exists a corresponding reduction in the Rho Calculus: if the term u reduces to the term v in a rewrite theory $\mathcal{R}$ we can build a ρ-term $\xi_{\mathcal{R}}(u)$ that reduces to the term $\{v\}$. The method used for constructing the term $\xi_{\mathcal{R}}(u)$ depends on all the reduction steps from u to v in the theory $\mathcal{R}$: $\xi_{\mathcal{R}}(u)$ is a representation in the Rho Calculus of the derivation trace. We can go further on and give a method for constructing a term $\xi_{\mathcal{R}}(u)$ without knowing a priori the derivation from u to v. Hence we want to answer to the following question: *"Given a rewrite theory $\mathcal{R}$, is there a ρ-term $\xi_{\mathcal{R}}$ such that for any term u if u normalizes to the term v in the rewrite theory $\mathcal{R}$ then $[\xi_{\mathcal{R}}](u)$ ρ-reduces to a set containing the term v?"* The definition of normalization strategies is in general done at the *meta-level* while the Rho Calculus allows us to represent such derivations at the *object level*.

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

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

In what follows we describe how the operators with the above functionalities can be defined in the Rho Calculus. We start with some auxiliary operators and afterwards, we introduce the ρ-operators that correspond to the functionalities listed above.

$$
\begin{array}{lrcl}
(First) & [\mathit{first}(s_1,\ldots,s_n)](t) & \to & \langle [s_1](t),\ldots,[s_n](t)\rangle \\
(FirstFail) & \langle \emptyset, t_1,\ldots,t_n\rangle & \to & \langle t_1,\ldots,t_n\rangle \\
(FirstSuccess) & \langle t, t_1,\ldots,t_n\rangle & \to & \{t\} \\
 & t \text{ contains no redexes, no free variables and is not } \emptyset & & \\
(FirstSingle) & \langle\rangle & \to & \emptyset
\end{array}
$$

Fig. 2. The *first* operator

3.2.1 Some Auxiliary Operators

First, we define three auxiliary operators that will be used in the next sections. These operators are just aliases used to define more complex ρ-terms and are used for giving more compact and clear definitions for the recursion operators. On the other hand, these operators correspond to the homonymous ones defined in ELAN.

The first of these operators is the *identity* (denoted id) that applied to any ρ-term t evaluates to the singleton containing this term,*i.e.* $[id](t) \mapsto_\rho \{t\}$:

$$id \stackrel{\triangle}{=} x \twoheadrightarrow x.$$

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

$$fail \stackrel{\triangle}{=} x \twoheadrightarrow \emptyset.$$

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

$$u; v \stackrel{\triangle}{=} x \twoheadrightarrow [v]([u](x)).$$

3.2.2 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 $\emptyset$. If all the arguments evaluate to $\emptyset$ then the final result of the evaluation is $\emptyset$. The evaluation rules describing the *first* operator and the auxiliary operator $\langle \_,\ldots,\_\rangle$ are presented in Figure 2. We will show later that this operator can be expressed in other versions of the rewriting calculus. The application of a ρ-term $\mathit{first}(s_1,\ldots,s_n)$ to a term t returns the result of the first "successful" application of one of its arguments to the term t. Hence, if $[s_i](t)$ evaluates to $\emptyset$ for $i = 1,\ldots,k-1$, and $[s_k](t)$ does not evaluate to $\emptyset$, then $[\mathit{first}(s_1,\ldots,s_n)](t)$ 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 $[\mathit{first}(s_1,\ldots,s_n)](t)$ does not terminate.

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

Fig. 3. Two traversal operators

Starting from the relation induced by the rules in Figures 1 and 2 we can define the classical notions of one-step ($\mapsto$), many-steps ($\twoheadmapsto_\rho$), and congruence ($=_\rho$) relation.

Example 3.2 The non-deterministic application of one of the rules $a \multimap b$, $a \multimap c$, $a \multimap d$ to the term a is represented in the Rho Calculus by the application $[\{a \multimap b, a \multimap c, a \multimap d\}](a)$. This last ρ-term is reduced to the term $\{b, c, d\}$ which represents a non-deterministic choice among the three terms. If we want to apply the above rules in a deterministic way and in the specified order, we use the ρ-term $[\mathit{first}(a \multimap b, a \multimap c, a \multimap d)](a)$ with, for example, the reduction:

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

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

3.2.3 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 Rho Calculus due to the rule *Cong* but we want to define a generic operator that applies a ρ-term r to the sub-terms u_i, $i = 1 \ldots n$, 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 3. These operators are inspired by the operators of the *System S* described in [24]. The application of the ρ-term $\Phi(r)$ to a term $t = f(u_1, \ldots, u_n)$ results in the successful application of the term r to one of the terms u_i. More precisely, r is applied to the first u_i, $i = 1, \ldots, n$ such that $[r](u_i)$ does not evaluate to the empty set. If there *exists no* such u_i and in particular, if t is a function with no arguments (t is a constant), then the term $[\Phi(r)](t)$ reduces to the empty set: $[\Phi(r)](c) \mapsto_{TraverseSeq} \langle\{\}\rangle \mapsto_{FirstFail} \langle\rangle \mapsto_{FirstSingle} \emptyset$. When the ρ-term

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

$$\Phi'(r) \triangleq \mathit{first}(f_1(r, id, \ldots, id), \ldots, f_1(id, \ldots, id, r), \ldots, f_m(r, id, \ldots, id), \ldots, f_m(id, \ldots, id, r))$$

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

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

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

and

$$[\Psi(r)](f_k(u_1, \ldots, u_p)) \mapsto\!\!\!\!\!\rightarrow_\rho \{f_k([r](u_1), \ldots, [r](u_p))\}$$

The operator Φ' does not correspond exactly to the definition from the Figure 3 but a similar result is obtained when applying the terms $\Phi(r)$ and $\Phi'(r)$ to a term $f_k(u_1, \ldots, u_p)$. We can thus state that the term traversal operators Φ and Ψ can be expressed in the Rho Calculus.

3.2.4 Iterators

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

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

We start by presenting the ρ-term used for describing recursive applications in the Rho Calculus. Starting from the Turing fixed-point combinator ([23]) we define the ρ-term $\Theta = [A](A)$ with

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

and for a given term G we obtain the reduction:

$$[\Theta](G) \mapsto_\rho \{[G]([\Theta](G))\} \qquad (FixPoint)$$

Since the ρ-term $[\Theta](G)$ can obviously lead to infinite reductions, a strategy should be used in order to obtain termination and thus the desired behavior. If Θ is considered as an independent ρ-term with the behavior described by an evaluation rule corresponding to the reduction $(FixPoint)$, the strategy suggested previously could be easily implemented. Alternatively, an *outermost* strategy can be used. It is clear that such a strategy prevents only the infinite reductions due to the operator Θ, but it cannot ensure the termination of the untyped Rho Calculus.

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

3.2.5 Top-down and Bottom-up Applications

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

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

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

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

$$\mathit{Once}_{bu}(r) \triangleq [\Theta](H_{bu}(r)).$$

The term $[\mathit{Once}_{bu}(r)](t) \triangleq [[\Theta](H_{bu}(r))](t)$ can lead to an infinite reduction if an appropriate strategy is not employed. Once again, we can use an outermost strategy in order to obtain the desired behavior.

Example 3.3 The application $[\mathit{Once}_{bu}(a \twoheadrightarrow b)](a)$ is reduced to the term $\{\langle [(a \twoheadrightarrow b)](a)\rangle\}$ and thus, to $\{b\}$. The application of the rule $a \twoheadrightarrow b$ to the leftmost-innermost position of a term $g(a, f(a))$ is represented by the term $[\mathit{Once}_{bu}(a \twoheadrightarrow b)](g(a, f(a)))$ and the corresponding evaluation is presented below:

$$\begin{aligned}
&[\mathit{Once}_{bu}(a \twoheadrightarrow b)](g(a, f(a))) \\
\mapsto_{\rho\!\!\rho} &\{\langle\langle g([\mathit{Once}_{bu}(a \twoheadrightarrow b)](a), f(a)), g(a, [\mathit{Once}_{bu}(a \twoheadrightarrow b)](f(a)))\rangle, [a \twoheadrightarrow b](g(a, f(a)))\rangle\} \\
\mapsto_{\rho\!\!\rho} &\{\langle\langle g(\{b\}, f(a)), g(a, [\mathit{Once}_{bu}(a \twoheadrightarrow b)](f(a)))\rangle, [a \twoheadrightarrow b](g(a, f(a)))\rangle\} \\
\mapsto_{\rho\!\!\rho} &\{\langle\{g(b, f(a))\}, [a \twoheadrightarrow b](g(a, f(a)))\rangle\} \\
\mapsto_{\rho\!\!\rho} &\{g(b, f(a))\}
\end{aligned}$$

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

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

$(Repeat_{success})$	$[repeat(r)](t)$	$\rightarrow$	$[repeat(r)]([r](t))$
			if $[r](t)$ is not reduced to $\emptyset$
$(Repeat_{fail})$	$[repeat(r)](t)$	$\rightarrow$	t
			if $[r](t)$ is reduced to $\emptyset$

Fig. 4. The operator *repeat*

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

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

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

3.2.6 Repetition and Normalization Operators

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

Now we want to define an operator that applies repeatedly a given strategy r to a ρ-term t. We call it *repeat* and its behavior can be described by the evaluation rules presented in Figure 4. 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 use once again the fixed-point operator presented in the previous section and we define the ρ-term

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

that is used for describing the *repeat* operator

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

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

Example 3.4 The repeated application of the rewrite rules $a \twoheadrightarrow b$ and $b \twoheadrightarrow c$ on the term a is represented by the term $[repeat(\{a \twoheadrightarrow b, b \twoheadrightarrow c\})](a)$ that evaluates as follows:

$$\begin{aligned}
&[repeat(\{a \twoheadrightarrow b, b \twoheadrightarrow c\})](a)\\
\mapsto_{\rho} &\{\langle [repeat(\{a \twoheadrightarrow b, b \twoheadrightarrow c\})]([\{a \twoheadrightarrow b, b \twoheadrightarrow c\}](a)), [id](a)\rangle\}\\
\mapsto_{\rho} &\{\langle [repeat(\{a \twoheadrightarrow b, b \twoheadrightarrow c\})](\{b\}), [id](a)\rangle\}\\
\mapsto_{\rho} &\{\langle\{\langle [repeat(\{a \twoheadrightarrow b, b \twoheadrightarrow c\})]([\{a \twoheadrightarrow b, b \twoheadrightarrow c\}](b)), [id](b)\rangle\}, [id](a)\rangle\}\\
\mapsto_{\rho} &\{\langle\{\langle [repeat(\{a \twoheadrightarrow b, b \twoheadrightarrow c\})](\{c\}), [id](b)\rangle\}, [id](a)\rangle\}
\end{aligned}$$

$\mapsto_{\rho} \{\langle\{\langle\{\langle[repeat(\{a \twoheadrightarrow b, b \twoheadrightarrow c\})](\{a \twoheadrightarrow b, b \twoheadrightarrow c\}](c)), [id](c)\rangle\}, [id](b)\rangle\}, [id](a)\rangle\}$
$\mapsto_{\rho} \{\langle\{\langle\{\langle([repeat(\{a \twoheadrightarrow b, b \twoheadrightarrow c\})](\emptyset), \{c\}\rangle\}, [id](b)\rangle\}, [id](a)\rangle\}$
$\mapsto_{\rho} \{\langle\{\langle\{\langle\langle\emptyset, \{c\}\rangle\rangle\}, [id](b)\rangle\}, [id](a)\rangle\}$
$\mapsto_{\rho} \{\langle\{\langle\{\langle\{c\}, [id](b)\rangle\}, [id](a)\rangle\}$
$\mapsto_{\rho} \{\langle\{\{\{c\}\}, [id](a)\rangle\}$
$\mapsto_{\rho} \{c\}$

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

$$im(r) \stackrel{\triangle}{=} repeat(Once_{bu}(r))$$

and an *outermost* strategy is defined by

$$om(r) \stackrel{\triangle}{=} repeat(Once_{td}(r)).$$

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

$$\xi_{\mathcal{R}}(u) \stackrel{\triangle}{=} [im(\mathcal{R})](u)$$

or

$$\xi_{\mathcal{R}}(u) \stackrel{\triangle}{=} [om(\mathcal{R})](u).$$

Example 3.5 If we denote by $\mathcal{R}$ the set of rules $\{a \twoheadrightarrow b, g(x, f(x)) \twoheadrightarrow x\}$, we represent by $[im(\mathcal{R})](g(a, f(a)))$ the leftmost-innermost normalization of the term $g(a, f(a))$ according to the set of rules $\mathcal{R}$ and the following derivation is obtained:

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

$\mapsto\!\!\!\!\rightarrow_p \{\langle\{\langle\{\{b\}\}, [id](g(b, f(a)))\rangle\}, [id](g(a, f(a)))\rangle\}$
$\mapsto\!\!\!\!\rightarrow_p \{\langle\{\{b\}\}, [id](g(a, f(a)))\rangle\}$
$\mapsto\!\!\!\!\rightarrow_p \{\langle\{b\}, [id](g(a, f(a)))\rangle\}$
$\mapsto\!\!\!\!\rightarrow_p \{b\}$

Given a term u, if the rewriting theory $\mathcal{R}$ is not confluent then, the result of the reduction of the term $[im(\mathcal{R})](u)$ is a set representing all the possible results of the reduction of the term u in the rewriting theory $\mathcal{R}$. Each of the elements of the result set represents the result of a reduction in the rewriting theory $\mathcal{R}$ for a given application order of the rewrite rules in $\mathcal{R}$.

Example 3.6 Let us consider the set $\mathcal{R} = \{a \twoheadrightarrow b, a \twoheadrightarrow c, g(x, x) \twoheadrightarrow x\}$ of non-confluent rewrite rules. The term $[im(\mathcal{R})](g(a, a))$ representing the *innermost* normalization of the term $g(a, a)$ according to the set of rewrite rules $\mathcal{R}$ is reduced to $\{b, g(c, b), g(b, c), c\}$. The term $[om(\mathcal{R})](g(a, a))$ representing the *outermost* normalization is reduced to $\{b, c\}$.

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

3.3 Encoding the "first" Operator

As we have seen in the previous section, the *first* operator plays a crucial role in the definition of the various strategies. One can wonder thus if this operator can be expressed using the basic operators of the Rho Calculus.

When trying to do this, the main difficulty is the ambivalent use of the empty set. For example, when applying on the term b the rewrite rule $a \twoheadrightarrow \emptyset$ that rewrites the constant a into the empty set, the evaluation rule of the calculus returns $\emptyset$ because *the matching against b fails*: $[a \twoheadrightarrow \emptyset](b) \mapsto_\rho \emptyset$. But it is also possible to explicitly rewrite an object into the empty set like in $[a \twoheadrightarrow \emptyset](a) \mapsto_\rho \emptyset$, and in this case the result is also the empty set because the rewrite rule *explicitly introduces* it.

It becomes then clear that one should avoid the ambivalent use of the empty set and introduce an *explicit distinction between failure and the empty set of results*. In fact, by making this distinction, we add to the matching power of the rewriting calculus a *catching power*, which allows to describe, in a rewriting style, exception mechanisms, and therefore to express easily convenient and elaborated strategies needed when computing or proving. An extended version of the Rho Calculus, denoted ρ_ε-calculus, was proposed in [15] where a single meaning is given to the empty set: to represent *only* the empty set of terms. Consequently, the application of a term to the empty set should

$(Fire)$	$[l \twoheadrightarrow r](t)$	$\rightarrow$	$\{\sigma r\}$
			where $\{\sigma\} = \mathcal{S}ol(l \prec t)$
$(Congr)$	$[f(t_1,\ldots,t_n)](f(u_1,\ldots,u_n))$	$\rightarrow$	$\{f([t_1](u_1),\ldots,[t_n](u_n))\}$
$(CongrFail)$	$[f(t_1,\ldots,t_n)](g(u_1,\ldots,u_n))$	$\rightarrow$	$\{\bot\}$
$(Distrib)$	$[\{u_1,\ldots,u_n\}](v)$	$\rightarrow$	if $n>0$, $\{[u_1](v),\ldots,[u_n](v)\}$
			if $n=0$, $\{\bot\}$
$(Batch)$	$[v](\{u_1,\ldots,u_n\})$	$\rightarrow$	if $n>0$, $\{[v](u_1),\ldots,[v](u_n)\}$
			if $n=0$, $\{\bot\}$
$(SwitchR)$	$u \twoheadrightarrow \{v_1,\ldots,v_n\}$	$\rightarrow$	$\{u \twoheadrightarrow v_1,\ldots,u \twoheadrightarrow v_n\}$
			if $n>0$
$(OpSet)$	$h(v_1,\ldots,\{u_1,\ldots,u_n\},\ldots,v_m)$	$\rightarrow$	$\{h(v_1,\ldots,u_i,\ldots,v_m)\}_{1\leqslant i\leqslant n}$
			if $n>0$ and $h\in\mathcal{F}_\varepsilon$
$(Flat)$	$\{u_1,\ldots,\{v_1,\ldots,v_m\},\ldots,u_n\}$	$\rightarrow$	$\{u_1,\ldots,v_1,\ldots,v_m,\ldots,u_n\}$
$(AppBotR)$	$[v](\bot)$	$\rightarrow$	$\{\bot\}$
$(AppBotL)$	$[\bot](v)$	$\rightarrow$	$\{\bot\}$
$(AbsBotR)$	$v \twoheadrightarrow \bot$	$\rightarrow$	$\{\bot\}$
$(OpBot)$	$f(t_1,\ldots,t_k,\bot,t_{k+1},\ldots,t_n)$	$\rightarrow$	$\{\bot\}$
$(FlatBot)$	$\{t_1,\ldots,\bot,\ldots,t_n\}$	$\rightarrow$	$\{t_1,\ldots,t_n\}$
			if $n>0$
(Exn)	$exn(t)$	$\rightarrow$	$\{t\}$
			$\{t\}\downarrow \neq \{\bot\}$ and $\mathcal{FV}(t\downarrow)=\emptyset$

Fig. 5. The evaluation rules of the ρ_ε-calculus

lead to failure (*i.e.* $[v](\emptyset) \mapsto_\rho \{\bot\}$) and the application of the empty set to a term should lead too to failure (*i.e.* $[\emptyset](v) \mapsto_\rho \{\bot\}$). On the other hand, provided all the t_i are in normal form, a term like $f(t_1,\ldots,\emptyset,\ldots,t_n)$ will be considered as a normal form and not rewritable to $\{\bot\}$. This is particularly useful to keep the first class status of the empty set. Moreover, we would like a ρ-term of the form $u \twoheadrightarrow \emptyset$ to be in normal form since it allows us to express a void function. The terms of the ρ_ε-calculus are defined by:

$$\mathcal{T} ::= \mathcal{X} \mid f(\mathcal{T},\ldots,\mathcal{T}) \mid \{\mathcal{T},\ldots,\mathcal{T}\} \mid [\mathcal{T}](\mathcal{T}) \mid \mathcal{T} \twoheadrightarrow \mathcal{T} \mid\bot\mid exn(\mathcal{T})$$

In ρ_ε-calculus, we generally consider only first-order terms in the left hand side of an abstraction. The patterns are thus built on variables and using symbols

from $\mathcal{F}_\varepsilon \triangleq \mathcal{F} \cup \{exn, \bot\}$. The evaluation rules of ρ_ε-calculus are described in Figure 5. For simplicity, in what follows we restrict to syntactic matching.

The ρ_ε-calculus is neither confluent nor strict. To obtain a confluent calculus, the evaluation mechanism must be tamed by a suitable strategy. In [15] this is done for example using call by value. Once a confluent strategy is defined, a *shallow* encoding of the *first* operator in the ρ_ε-calculus can be given, *i.e.* we can define a term to express it.

The role of the *first* operator is to select between its arguments the first one that, applied to a given ρ-term, does not evaluate to $\{\bot\}$. This is something quite difficult to express in the Rho Calculus. We propose to represent the term $[\mathit{first}(t_1, \ldots, t_n)](r)$ by a set of n terms, denoted $\{u_1, \ldots, u_n\}$, with the property that every term u_i can be reduced to $\{\bot\}$ except perhaps one (say u_l). In other words, the initial set is reduced to a singleton, containing the normal from of u_l, by successively reducing each u_i $(i \neq l)$ to $\{\bot\}$ and then using the $(FlatBot)$ rule. To express *first* in the ρ_ε-calculus, we can use its two main trumps: the matching and the failure catching. Each u_i is expressed using the $i-1$ first terms: if all u_j $(j < i)$ are reduced to $\{\bot\}$, then u_i leads to $[t_i](r)$ else u_i leads to $\{\bot\}$ by a rule application failure. We define u_i as:

$$
\begin{aligned}
u_1 &\triangleq [t_1](x)\\
u_2 &\triangleq \big[exn(\bot) \twoheadrightarrow [t_2](x)\big]\big(exn(u_1)\big)\\
u_3 &\triangleq \big[exn(\bot) \twoheadrightarrow [exn(\bot) \twoheadrightarrow [t_3](x)](exn(u_2))\big]\big(exn(u_1)\big)\\
u_4 &\triangleq \big[exn(\bot) \twoheadrightarrow [exn(\bot) \twoheadrightarrow [exn(\bot) \twoheadrightarrow [t_4](x)](exn(u_3))](exn(u_2))\big]\big(exn(u_1)\big)\\
&\vdots\\
u_n &\triangleq [exn(\bot) \twoheadrightarrow [exn(\bot) \twoheadrightarrow [\ldots \twoheadrightarrow [t_n](x)](exn(u_{n-1}))\ldots](exn(u_2))](exn(u_1))
\end{aligned}
$$

and we define *first* by:

$$\mathit{first}(t_1, \ldots, t_n) \triangleq x \twoheadrightarrow \{u_1, \ldots, u_n\}.$$

Example 3.7 Using this *first* operator and the failure catching mechanism, we can express the evaluation scheme:

if M is evaluated to the failure term
then evaluate the term P_{fail}
else evaluate the term P_{nor}.

thanks, for example, to the term $[\mathit{first}(exn(\bot) \twoheadrightarrow P_{fail}, x \twoheadrightarrow P_{nor})](exn(M))$.

(i) if $M \mapsto_{\rho_\varepsilon} \{\bot\}$, we have the following reduction:

$$
\begin{aligned}
&\big[\mathit{first}\big(exn(\bot) \twoheadrightarrow P_{fail}, x \twoheadrightarrow P_{nor}\big)\big]\big(exn(M)\big)\\
\mapsto_{\rho_\varepsilon}\ &\big[\mathit{first}\big(exn(\bot) \twoheadrightarrow P_{fail}, x \twoheadrightarrow P_{nor}\big)\big]\big(exn(\{\bot\})\big)\\
\mapsto_{OpSet}\ &\big[\mathit{first}\big(exn(\bot) \twoheadrightarrow P_{fail}, x \twoheadrightarrow P_{nor}\big)\big]\big(\{exn(\bot)\}\big)
\end{aligned}
$$

$$\mapsto_{\rho_\varepsilon} \quad \{P_{fail}\}$$

and thus, the initial term leads to P_{fail}, which is the wanted behavior.

(ii) if $M \mapsto_{\rho_\varepsilon} \{M' \downarrow\}$ where $\{M' \downarrow\} \neq \{\bot\}$ is in normal form, we obtain:

$$\begin{array}{ll} & \Big[\mathit{first}\big(\mathit{exn}(\bot) \twoheadrightarrow P_{fail}, x \twoheadrightarrow P_{nor}\big)\Big]\big(\mathit{exn}(M)\big) \\ \mapsto_{\rho_\varepsilon} & \Big[\mathit{first}\big(\mathit{exn}(\bot) \twoheadrightarrow P_{fail}, x \twoheadrightarrow P_{nor}\big)\Big]\big(\mathit{exn}(M')\big) \\ \mapsto_{\rho_\varepsilon} & \{P_{nor}\} \end{array}$$

and in this case also, we have the wanted result: P_{nor}.

3.4 Back to ELAN

The rules of the system ELAN can be expressed using the Rho Calculus. A rule with no conditions and no local assignments $l \Rightarrow r$ is represented by $l \twoheadrightarrow r$. In fact, the ELAN evaluation mechanism distinguishes 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 and thus, each time a labeled rewrite rule is applied to a term, the ELAN evaluation mechanism normalizes the result of its application with respect to the set of unlabeled rewrite rules. Hence, the labeled rewrite rule $[lab]\ \ l \Rightarrow r$ is represented by $l \twoheadrightarrow [\mathit{im}(\mathcal{R})](r)$ where $\mathcal{R}$ is the representation of the set of unlabeled ELAN rules.

The elementary ELAN strategies have, in most of the cases, a direct representation in the Rho Calculus. The identity (id) and the failure (fail) as well as the concatenation (;) are directly represented in the Rho Calculus by the ρ-operators id, $fail$ and ";" respectively, defined in Section 3.2.1. The strategy $\mathsf{dk}(S_1, \ldots, S_n)$ is represented in the Rho Calculus by the set $\{S_1, \ldots, S_n\}$ and the strategy $\mathsf{first}(S_1, \ldots, S_n)$ by the ρ-term $\mathit{first}(S_1, \ldots, S_n)$. The iteration strategy operator repeat* is easily represented by using the ρ-operator *repeat*.

The ELAN strategies are expressed using rewrite rules and therefore, can be represented by ρ-terms in the same way as the ELAN rewrite rules.

We have briefly described here the representation of simple ELAN rules and the corresponding implicit and user defined strategies of ELAN. This representation can be extended to more general rules with conditions and local evaluations as detailed in [8].

4 Enhancing the Rewriting Calculus Design

In this section we present the untyped syntax of the rewriting calculus as it was first given in [10] and that enhances (*i.e.* get simpler but as expressive as before) the one presented in the previous section. In what follows, we will quickly introduce the new syntax of this formalism that we denote ρCal, two operational semantics (small-step and big-step) and we will present different evaluation strategies. In particular we show how a small enhancement in the calculus (more precisely in the big-step) semantics could help us to encode the

first operator.

4.1 The Syntax of the ρCal

The syntax of the enhanced ρCal is defined as follows:

$$\mathcal{T} ::= \mathcal{X} \mid \mathcal{K} \mid \mathcal{T} \twoheadrightarrow \mathcal{T} \mid [\mathcal{T} \ll \mathcal{T}].\mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{T}, \mathcal{T}$$

where $\mathcal{X}$ represents a denumerable set of variables and $\mathcal{K}$ a set of constants. By abuse of notation, we denote members of these sets by the same letters possibly indexed. The characteristics of this new syntax are the following:

(i) $\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2$ denotes a *rule abstraction* with pattern $\mathcal{T}_1$ and body $\mathcal{T}_2$; this is unchanged with respect to the initial version.

(ii) $[\mathcal{T}_1 \ll \mathcal{T}_2].\mathcal{T}_3$ denotes a *delayed matching constraint*; this is a major evolution in the syntax and capability of the calculus. The application of an abstraction $\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_3$ to a term $\mathcal{T}_2$ always "fires" and produces the term $[\mathcal{T}_1 \ll \mathcal{T}_2].\mathcal{T}_3$ which represents a constrained term where the matching equation is "put on the stack". If a solution σ of the matching between $\mathcal{T}_1$ and $\mathcal{T}_2$ exists, the delayed matching constraint can be evaluated to $\sigma(\mathcal{T}_3)$.

(iii) The application operator, previously denoted $[\mathcal{T}_1](\mathcal{T}_2)$, is now simply denoted as in the lambda calculus $(\mathcal{T}_1\ \mathcal{T}_2)$.

(iv) Finally, the use of the set brackets in the initial syntax can be usefully decomposed into two quite different parts. The first is a structure constructor denoted ",". The second is the semantics given to the structure constructor, which is described by an appropriate theory that depends on the kind of result one wants to formalize. Typically we recover the initial semantics of sets of results by giving an associative-commutative and idempotent semantics to ",". If one prefers lists or multisets results, then the corresponding formalization of "," should be specified.

As in the first version of the calculus, the substitutions are higher-order substitutions but grafting can be used when working modulo α-*conversion*. Similarly, the same notion of matching as presented in Section 3.1 is used but we focus here on *syntactic matching*.

The small-step reduction semantics is defined by the reduction rules presented below: The central idea of the (ρ) rule of the calculus is that the application of a term $\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2$ to a term $\mathcal{T}_3$ reduces to the delayed matching constraint $[\mathcal{T}_1 \ll \mathcal{T}_3].\mathcal{T}_2$, while (the application of) the (σ) rule consists in solving the matching equation $\mathcal{T}_1 \prec\!\!\!\prec \mathcal{T}_3$, and applying the obtained result to the the term $\mathcal{T}_2$. The rule (δ) deals with the distributivity of the application on the structures built with the "," constructor. As usual, we can introduce the classical notions of one-step, many-steps ($\mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta}$), and congruence ($=_{\rho\sigma\delta}$) relation *w.r.t.* $\rightarrow_{\rho\sigma\delta}$.

$$
\begin{array}{llll}
(\rho) & (\mathcal{T}_1 \twoheadrightarrow \mathcal{T}_2)\mathcal{T}_3 & \rightarrow_\rho & [\mathcal{T}_1 \ll \mathcal{T}_3].\mathcal{T}_2 \\
(\sigma) & [\mathcal{T}_1 \ll \mathcal{T}_3].\mathcal{T}_2 & \rightarrow_\sigma & \sigma_{(\mathcal{T}_1 \prec \mathcal{T}_3)}(\mathcal{T}_2) \\
(\delta) & (\mathcal{T}_1, \mathcal{T}_2)\,\mathcal{T}_3 & \rightarrow_\delta & \mathcal{T}_1\,\mathcal{T}_3, \mathcal{T}_2\,\mathcal{T}_3
\end{array}
$$

Fig. 6. The small-step semantics of the ρCal

4.2 Well-typed Fix-point

Various typed versions of the ρCal have been or are being developed, for different purposes: normalization, automated deduction, typed programming disciplines, *etc.*

Here we present a simple (first-order) type inference system *à la* Curry which affects types to some terms of the ρCal. The rules are adapted from the polymorphic type system for the ρCal [10], and the full first-order type system *à la* Church for the ρCal can be found in [11].

In a nutshell, monomorphic and polymorphic type systems for ρCal have the "nice" property that they allow the encoding of many fix-points. Therefore, they are suitable to be taken as a foundational basis of realistic rewriting- and functional-based programming languages. The type system is adapted from the simply typed λ-calculus by generalizing the abstraction rule for patterns:

$$
\frac{\Gamma, \Delta \vdash \mathcal{P} : \phi \qquad \Gamma, \Delta \vdash \mathcal{T}_2 : \psi \quad \mathsf{Dom}(\Delta) = \mathsf{Fv}(\mathcal{P})}{\Gamma \vdash \mathcal{P} \twoheadrightarrow_\Delta \mathcal{T}_2 : \phi \twoheadrightarrow \psi}\ (Abs) \qquad \frac{\Gamma \vdash \mathcal{T}_1 : \phi \twoheadrightarrow \psi \quad \Gamma \vdash \mathcal{T}_2 : \phi}{\Gamma \vdash \mathcal{T}_1\ \mathcal{T}_2 : \psi}\ (Appl)
$$

As a simple example, we present here a term inspired by the famous $\omega\,\omega$ term of the untyped λ-calculus. Basically, the term $X \twoheadrightarrow X\,X$ can not be typed because X can not have type α and $\alpha \twoheadrightarrow \alpha$ at the same time. However, using the constant f whose type is $(\alpha \twoheadrightarrow \alpha) \twoheadrightarrow \alpha$, we can switch between these types. Then we define:

$$\omega_f \stackrel{\triangle}{=} f(X) \twoheadrightarrow_{(X:\alpha \twoheadrightarrow \alpha)} X\ f(X)$$

and we can typecheck it the following way, assuming the type of constant to be given in a suitable signature omitted here but clear from the context:

$$
\frac{\dfrac{(1)}{\vdash \omega_f : \alpha \twoheadrightarrow \alpha} \qquad \dfrac{\vdash f : (\alpha \twoheadrightarrow \alpha) \twoheadrightarrow \alpha \quad \vdash \omega_f : \alpha \twoheadrightarrow \alpha}{\vdash f(\omega_f) : \alpha}}{\vdash \omega_f\ f(\omega_f) : \alpha}
$$

where (1) is:

$$\dfrac{\dfrac{X{:}\alpha \twoheadrightarrow \alpha \vdash f : (\alpha \twoheadrightarrow \alpha) \twoheadrightarrow \alpha \qquad X{:}\alpha \twoheadrightarrow \alpha \vdash X : \alpha \twoheadrightarrow \alpha}{X{:}\alpha \twoheadrightarrow \alpha \vdash f(X) : \alpha} \qquad \dfrac{X{:}\alpha \twoheadrightarrow \alpha \vdash X : \alpha \twoheadrightarrow \alpha \qquad X{:}\alpha \twoheadrightarrow \alpha \vdash f(X) : \alpha}{X{:}\alpha \twoheadrightarrow \alpha \vdash X\ f(X) : \alpha}}{\vdash \omega_f : \alpha \twoheadrightarrow \alpha}$$

Finally, the only reduction path from $\omega_f\ f(\omega_f)$ can not end:

$$\begin{aligned} \omega_f\ f(\omega_f) \ &\equiv\ (f(X) \twoheadrightarrow_{(X:\alpha\twoheadrightarrow\alpha)} X\ f(X))\ f(\omega_f) \\ &\mapsto_{\rho}\ [f(X) \ll_{(X:\alpha\twoheadrightarrow\alpha)} f(\omega_f)].(X\ f(X)) \\ &\mapsto_{\sigma}\ (X\ f(X))[\omega_f/X] \equiv \omega_f\ f(\omega_f) \\ &\mapsto_{\rho} \ldots \end{aligned}$$

This kind of typed fix-point is made possible because any well-formed type can be chosen for the constants of the calculus. Here, f has type $(\alpha \twoheadrightarrow \alpha) \twoheadrightarrow \alpha$, so at the type level it makes a function over the set α look like an object in α.

A similar method can be used in ML in order to build a fix-point without using `let rec`. Instead of defining a constant with an arbitrary type, we just have to define a peculiar inductive type whose unique constructor behaves like f:

```
type t = F of (t -> t);;
let omega x = match x with (F y) -> y (F y);;
```

Then the evaluation of `omega (F omega)` does not terminate.

In the ρCal, we have the possibility to use explicitly this kind of constants every time we want to express recursion. By checking the type of the constants and the context they are used in, this gives us the possibility to have an accurate control over non-termination.

4.3 Encoding TRS in ρCal

In what follows, we show how to build some elaborated terms describing the application of a rewrite system on a term according to a specific strategy. We do not give the type derivations, but all the terms given here remain typable with the same system as in the previous subsection. Since we will have to "try" all the rules of the rewrite system on a given subterm, the reductions in our terms will generate a lot of "junk" terms corresponding to the various matching failures. We will deal with them thanks to an enhanced equivalence theory on terms which eliminates definitively-stuck-values, obtained by non recoverable matching failures. We call this theory $\mathbb{T}_{\mathsf{nostuck}}$ and use it for defining equivalence classes over the structure operator ",", in order to properly

define a "result" term:

$$\frac{\nexists \theta_1, \theta_2,\ \theta_1(\mathcal{T}_1) \mapsto_{\rho\partial\delta} \theta_2(\mathcal{P})}{[\mathcal{P} \ll \mathcal{T}_1].\mathcal{T}_2, \mathcal{T}_3 =^{\mathsf{no}}_{\mathsf{stuck}} \mathcal{T}_3} \quad (\mathbb{T}_{\mathsf{nostuck}})$$

Intuitively, this theory drops, in a structure, all the matching failures which can not be made solvable by a further instantiation or reduction. For instance, $[f(X) \ll g(3)].4$ will be dropped, but not $[f(3) \ll f(X)].4$ (this last one becoming solvable if X is instantiated to 3). Note that, since the patterns contain no structures, the matching remains syntactic. The $=^{\mathsf{no}}_{\mathsf{stuck}}$ equivalence is clearly not decidable and an approximation should be defined.

Starting from the encoding of object calculi [9] we define a suitable object-based recursion operator that allows us to simulate the *global* behavior of a TRS $\mathcal{R}$.

We begin with the example of the addition defined by the following TRS:

$$\begin{aligned} add(0, Y) &\to Y \\ add(suc(X), Y) &\to suc\,(add(X, Y)) \end{aligned}$$

Using the auxiliary constant rec we define the following ρ-term that computes the addition over Peano integers:

$$plus \triangleq \begin{pmatrix} rec(S) \twoheadrightarrow add(0, Y) \twoheadrightarrow Y, \\ rec(S) \twoheadrightarrow add(suc(X), Y) \twoheadrightarrow suc\,(S.rec \;\; add(X, Y)) \end{pmatrix}$$

If we consider the expressions "$\overline{m}$", and "$\overline{m+n}$", and "$\overline{m-n}$" as aliases for the Peano representations of these numbers as sequences of $suc(\ldots suc(0)\ldots)$) the reductions below are obtained. To ease the reading, within the failed matching constraints, we keep only the subterms which do lead to a failure (for instance, in $[add(0, Y) \ll add(\overline{n}, \overline{m})]$, we only keep $[0 \ll \overline{n}]$):

$$\begin{aligned} & plus.rec \;\; add(\overline{n}, \overline{m}) \\ \mapsto_{\rho\partial\delta} \;& (add(0, Y) \twoheadrightarrow Y) \;\; add(\overline{n}, \overline{m}), \\ & (add(suc(X), Y) \twoheadrightarrow suc\,(plus.rec \;\; add(X, Y))) \;\; add(\overline{n}, \overline{m}) \\ & \ldots \\ \mapsto_{\rho\partial\delta} \;& [0 \ll \overline{n}].\overline{m}\ , \quad [0 \ll \overline{n-1}].(\overline{m+1}), \cdots \\ & [0 \ll 0].(\overline{m+n}), \\ & [suc(X) \ll 0].(suc\,(plus.rec \;\; add(X, Y))) \\ =^{\mathsf{no}}_{\mathsf{stuck}} \;& [0 \ll 0].(\overline{m+n}) \\ \mapsto_{\sigma} \;& \overline{m+n} \end{aligned}$$

Worthy noticing is that all the stuck results are dropped by $=^{\mathsf{no}}_{\mathsf{stuck}}$; the only interesting member of the structure is $[0 \ll 0].(\overline{m+n})$. On the left of this term, all the terms get stuck because we try to match 0 against $suc(\overline{n})$; on the right too because we try to match $suc(X)$ against 0.

This mechanical encoding works for many TRS as well: one just has to put the subterm "$S.rec$ " before all the defined constants, so that the whole rewrite system can be re-applied to any of the corresponding subterms.

Let us see a slightly more elaborated example:

$$fibonacci \triangleq \left(\begin{array}{lll} rec(S) \twoheadrightarrow fibo(0) & \twoheadrightarrow 1 & , \\ rec(S) \twoheadrightarrow fibo(suc(0)) & \twoheadrightarrow 1 & , \\ rec(S) \twoheadrightarrow fibo(suc(suc(N))) & \twoheadrightarrow plus.rec\ add(S.rec\ fibo(N), & \\ & \quad S.rec\ fibo(suc(N))) & \end{array} \right)$$

The engine we have shown has a *lazy innermost* policy: because of the presence of $S.rec$ before all the defined symbols, a position which is not innermost in the term can be reduced if and only if the corresponding rewrite rule discards all the subterms which contain a defined constant.

Other versions of this engine can be designed. For example, the ρ-term given below computes the length of a list with an *outermost* evaluation strategy. The first call should have shape $length.rec\ \ len(0, L)$.

$$length \triangleq \left(\begin{array}{l} rec(S) \twoheadrightarrow len(N, nil) \twoheadrightarrow N, \\ rec(S) \twoheadrightarrow len(N, cons(X, L)) \twoheadrightarrow S.rec\ \ len(suc(N), L) \end{array} \right)$$

This kind of encoding works well for rewrite systems written in a Prolog style, since there is always a defined symbol (len) on the head of the term, and the result is accumulated in an argument (N). As in Prolog too, programming with accumulators is very efficient because it roughly corresponds to tail recursion, avoiding a series of useless matchings at the end of the computation.

To end up with traversal, let us have a glance at how one can define a "customized" evaluation strategy. As in ELAN, a user of the ρCal may want to combine different strategies in the same rewrite system. For instance, in a function which selects a part of a list, the traversal of the list can be done straightforwardly (to generate less "junk" terms) and a lazy innermost strategy is better for the evaluation of each member of the list (because it is more efficient).

The easiest way to obtain this kind of user-defined evaluation is to have one term for each specific strategy and call them mutually. Let us take the example of a function which selects all the numbers greater than two in a list. On the one hand, we will have a traversal term which browses the list,

distributing the comparison to 2 on every member:

$$select \triangleq \left(\begin{array}{l} rec2(T) \to nil \to nil, \\ rec2(T) \to cons(0, L) \to T.rec2\ L, \\ rec2(T) \to cons(suc(0), L) \to T.rec2\ L, \\ rec2(T) \to cons(suc(suc(N)), L) \to cons(suc(suc(N)), T.rec2\ L) \end{array} \right)$$

On the other hand, the members of the list can be written with *plus* (as before), so their evaluation is innermost. This encoding will be efficient because *plus* "pushes" the constants *suc* directly out of the term. Thus, the comparison to 2 only requires partial evaluation of the members of the list, so that the TRS *select* can check whether there are two "*suc*" or not at the beginning of these members. For instance, we have the following computation:

$$\begin{array}{l} \quad select.rec2\, cons(\overline{5} + \overline{1}, cons(\overline{1} + \overline{0}, cons(\overline{1} + \overline{4}, nil))) \\ \mapsto\!\!\!\!\!\!\rightarrow_{\mathcal{P}os} cons(suc(suc(\overline{3} + \overline{1})), cons(\overline{5}, nil)) \end{array}$$

Notice that we enforce an evaluation mechanism similar to the one in ELAN, because *plus* carries in fact the whole TRS defining addition (which can be considered as unlabeled), together with its associated innermost strategy.

4.4 Encoding "first" Using Natural Semantics

We define an operational semantics via a natural proof deduction system *à la* Kahn [18]. The purpose of the deduction system is to map every closed expression into a normal form, *i.e.* an irreducible term in weak head normal form. The presented strategy is *lazy call-by-name* since it does not work under plain abstractions (*i.e.* $\mathcal{T} \to \mathcal{T}$), structures (*i.e.* $\mathcal{T}, \mathcal{T}$), and algebraic terms (*i.e.* $\mathcal{K}\,\mathcal{T}$). We define the set of values $\mathcal{V}$ and output values $\mathcal{O}$ as follows:

$$\mathcal{V} ::= \mathcal{K} \mid \mathcal{T} \to \mathcal{T} \mid \mathcal{K}\,\mathcal{T} \mid \mathcal{T}, \mathcal{T}$$

$$\mathcal{O} ::= \mathcal{V} \mid \mathsf{wrong}$$

The special output wrong represents the result obtained by a computation involving a "matching equation failure" (represented in [9] by null). The semantics is defined via a judgment of the shape $\mathcal{T} \Downarrow \mathcal{O}$, and its rules (almost self explaining) are presented in Figure 7. The big-step operational semantics is deterministic, and immediately suggests how to build an interpreter for the calculus.

Example 4.1 [A call-by-value derivation] Let $\omega\,\omega \triangleq (\mathcal{X} \to \mathcal{X}\,\mathcal{X})\,(\mathcal{X} \to \mathcal{X}\,\mathcal{X})$, and take the term $(\mathcal{X} \to 3)\,(\omega\,\omega)$. One can check that this term diverges using

$$\frac{}{\mathcal{V} \Downarrow \mathcal{V}} \quad (Red{-}Val)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3 \twoheadrightarrow \mathcal{T}_4 \quad [\mathcal{T}_3 \ll \mathcal{T}_2].\mathcal{T}_4 \Downarrow \mathcal{O}}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow \mathcal{O}} \quad (Red{-}\rho)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3, \mathcal{T}_4 \quad \mathcal{T}_3\,\mathcal{T}_2 \Downarrow \mathcal{V}_1 \quad \mathcal{T}_4\,\mathcal{T}_2 \Downarrow \mathcal{V}_2}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow \mathcal{V}_1, \mathcal{V}_2} \quad (Red{-}\delta)$$

$$\frac{\exists \sigma.\ \sigma(\mathcal{T}_1) \equiv \mathcal{T}_2 \quad \sigma(\mathcal{T}_3) \Downarrow \mathcal{O}}{[\mathcal{T}_1 \ll \mathcal{T}_2].\mathcal{T}_3 \Downarrow \mathcal{O}} \quad (Red{-}\sigma_1)$$

$$\frac{\nexists \sigma.\ \sigma(\mathcal{T}_1) \equiv \mathcal{T}_2}{[\mathcal{T}_1 \ll \mathcal{T}_2].\mathcal{T}_3 \Downarrow \mathsf{wrong}} \quad (Red{-}\sigma_2)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathsf{wrong}}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow \mathsf{wrong}} \quad (Red{-}Prop_1)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3, \mathcal{T}_4 \quad \mathcal{T}_3\,\mathcal{T}_2 \Downarrow \mathsf{wrong}}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow \mathsf{wrong}} \quad (Red{-}Prop_2)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{T}_3, \mathcal{T}_4 \quad \mathcal{T}_3\,\mathcal{T}_2 \Downarrow \mathcal{V} \quad \mathcal{T}_4\,\mathcal{T}_2 \Downarrow \mathsf{wrong}}{\mathcal{T}_1\,\mathcal{T}_2 \Downarrow \mathsf{wrong}} \quad (Red{-}Prop_3)$$

Fig. 7. The natural semantics of the ρCal

a call-by-value strategy, schematically:

$$\frac{\begin{array}{c}\text{“}\infty\text{”}\\ \therefore\end{array}}{(\mathcal{X} \twoheadrightarrow 3)\,(\omega\,\omega) \Downarrow \text{“}\mathsf{stack\ overflow}\text{”}}$$

while it would converge to 3 using call-by-name.

It is worth noticing that the output value wrong is used in order to denote “bad” computations where matching failure occurs at run-time. As such, the presented semantics does abort computations, as for example in

$$\frac{\vdots}{(3 \twoheadrightarrow 3, 4 \twoheadrightarrow 4)\,4 \Downarrow \mathsf{wrong}}$$

keeping in the final result the fact that one computation goes wrong and throwing away all non-wrong results. This corresponds to “killing” the com-

putation once a wrong value is produced; as such, our machine is "pessimistic" (the machine stops if at least one wrong occurs).

In the following, we present a small extension of ρCal and its big-step semantics which takes into account a comfortable encoding of the *first* operator. To do this we need first to add an "exception" handling constructor to the ρCal syntax, *i.e.*

$$\mathcal{T} ::= \mathsf{try}\ \mathcal{T}\ \mathsf{with}\ \mathcal{T} \mid \ldots \text{ as before } \ldots$$

Executing a $\mathsf{try}\ \mathcal{T}_1\ \mathsf{with}\ \mathcal{T}_2$ means that if a matching failure occurs when evaluating the term $\mathcal{T}_1$, the handler $\mathcal{T}_2$ will be executed. Otherwise, the result of the execution of $\mathcal{T}_1$ will be propagated outside the scope of the try_with.

More precisely: we first evaluate $\mathcal{T}_1$ (the protected term) and if $\mathcal{T}_1$ evaluates to an output without matching failures (*i.e.* an output value), then this value will be the result of the whole try_with expression. Conversely, if a matching failure occurs, then we execute the handler $\mathcal{T}_2$. Note that in the case of a multiple nesting of try_with, our operational semantics will handle the feature by executing the innermost "handler". This simple extension allows us to easily encode the *first* operator.

Therefore, we keep all the previous rules and we add the two deduction rules below:

$$\frac{\mathcal{T}_1 \Downarrow \mathsf{wrong} \quad \mathcal{T}_2 \Downarrow \mathcal{O}}{\mathsf{try}\ \mathcal{T}_1\ \mathsf{with}\ \mathcal{T}_2 \Downarrow \mathcal{O}}\ (Red{-}Fail_1)$$

$$\frac{\mathcal{T}_1 \Downarrow \mathcal{V}}{\mathsf{try}\ \mathcal{T}_1\ \mathsf{with}\ \mathcal{T}_2 \Downarrow \mathcal{V}}\ (Red{-}Fail_2)$$

Intuitively, the first deduction rule evaluates the handler $\mathcal{T}_2$ if and only if the protected body $\mathcal{T}_1$ reduces to wrong; otherwise the protected body $\mathcal{T}_1$ succeeds without matching failures, and the second rule applies.

As before, the presented interpreter implements a "pessimistic" machine that strictly propagates the matching failure signal. Thus, one should remember that, according to the propagation rules, a matching failure signal is propagated independently of the other (possibly successful) computations.

We are now ready to provide a canonical encoding of the *first* operator using the newly introduced try_with construct as follows:

$$\mathit{first}(\mathcal{T}_1 \cdots \mathcal{T}_n) \triangleq \mathcal{X} \twoheadrightarrow (\mathsf{try}\ \mathcal{T}_1\,\mathcal{X}\ \mathsf{with}\ \mathsf{try}\ \mathcal{T}_2\,\mathcal{X}\ \mathsf{with}\ \ldots\ \mathsf{try}\ \mathcal{T}_{n-1}\,\mathcal{X}\ \mathsf{with}\ \mathcal{T}_n\,\mathcal{X})$$

Example 4.2 [One run of *first*] Take the term $\mathit{first}(1 \twoheadrightarrow 1, 2 \twoheadrightarrow 2, 3 \twoheadrightarrow 6)\,2$.

A derivation for this term is:

$$\dfrac{\dfrac{\dfrac{}{1 \twoheadrightarrow 1 \Downarrow 1 \twoheadrightarrow 1} \quad \dfrac{\nexists \sigma.\sigma(1) \equiv 2}{[1 \ll 2].1 \Downarrow \mathsf{wrong}}}{(1 \twoheadrightarrow 1)\,2 \Downarrow \mathsf{wrong}} \quad \dfrac{\dfrac{\dfrac{}{2 \twoheadrightarrow 2 \Downarrow 2 \twoheadrightarrow 2} \quad \dfrac{\exists \sigma_{\mathsf{ID}}.\sigma_{\mathsf{ID}}(2) \equiv 2 \quad 2 \Downarrow 2}{[2 \ll 2].2 \Downarrow 2}}{(2 \twoheadrightarrow 2)\,2 \Downarrow 2}}{\mathsf{try}\ (2 \twoheadrightarrow 2)\,2\ \mathsf{with}\ (3 \twoheadrightarrow 6)\,2 \Downarrow 2}}{\mathsf{try}\ (1 \twoheadrightarrow 1)\,2\ \mathsf{with\ try}\ (2 \twoheadrightarrow 2)\,2\ \mathsf{with}\ (3 \twoheadrightarrow 6)\,2 \Downarrow 2}$$
$$\overline{\mathit{first}(1 \twoheadrightarrow 1, 2 \twoheadrightarrow 2, 3 \twoheadrightarrow 6)\,2 \Downarrow 2}$$

4.5 Encoding First-class Exceptions

In the previous section, we saw that the wrong output value aborts computations, and the construct try_with was introduced in order to encode the *first* operator. Nevertheless, modern programming languages are interested in trying some chunks of code in a protected environment, and whenever a run-time matching error occurs, catch it first, and then execute some exception handler which can be declared many "miles" away from where the exception was raised. This is the case of the `try{...}catch{...}` mechanism of Java, or similar constructs in ML, C#, ... In ρCal an exception can be represented by the fact that some matching failure occurred at run-time, such as matching the constant 3 against another constant 4.

Among the possible extensions of ρCal taking into account first-class exceptions it is worth to mention:

(i) In [10], we presented an extension of ρCal which takes into account matching exceptions and their handling. This extension was also inspired from [21]. We extended the syntax as follows:

$$\mathcal{T} ::= \mathsf{try}\ \mathcal{T}\ \mathsf{catch}\ [\mathcal{T} \ll \mathcal{T}]\ \mathsf{with}\ \mathcal{T} \mid \ldots \text{ as before } \ldots$$

Intuitively, $[\mathcal{T} \ll \mathcal{T}]$ denotes a matching equation without solutions, like *e.g.* $[3 \ll 4]$. Executing a $\mathsf{try}\ \mathcal{T}_1\ \mathsf{catch}\ [\mathcal{T}_2 \ll \mathcal{T}_3]\ \mathsf{with}\ \mathcal{T}_4$ means that the scope for catching the matching failure $[\mathcal{T}_2 \ll \mathcal{T}_3]$ is the term $\mathcal{T}_1$, and that if that exception occurs, the handler $\mathcal{T}_4$ will be executed. Otherwise the raised exception will be propagated outside the scope of the try_catch_with.

More precisely, we first evaluate $\mathcal{T}_1$ (the protected term) and if $\mathcal{T}_1$ evaluates to an output without matching failures (*i.e.* a value), then this value will be the result of the whole try_catch_with expression (this roughly corresponds to executing $\mathcal{T}_1$ without raising exceptions). Conversely, if a matching failure occurs, then we must check whether the failure is the one declared in the try_catch_with expression (*i.e.* $[\mathcal{T}_2 \ll \mathcal{T}_3]$) or not; in the first case we execute the handler $\mathcal{T}_4$, while in the latter case we propagate the matching failure outside the scope of the try_catch_with expression. Note that the scope of the exception catching ranges over $\mathcal{T}_1$ but not $\mathcal{T}_4$.

(ii) We are currently studying another possible choice for handling exceptions where the syntax is extended as follows:

$$\mathcal{T} ::= \mathsf{try}\ \mathcal{T}\ \mathsf{with}\ \mathcal{T} \twoheadrightarrow \mathcal{T} \mid \mathsf{raise}\ \mathcal{T} \mid \ldots \text{ as before } \ldots$$

The term $\mathsf{try}\ \mathcal{T}_1\ \mathsf{with}\ \mathcal{T}_2 \twoheadrightarrow \mathcal{T}_3$ represents the evaluation of the term $\mathcal{T}_1$ in a protected environment. As in the previous approach, we first evaluate $\mathcal{T}_1$ and if $\mathcal{T}_1$ evaluates to an output without matching failures, then this value will be the result of the whole try_with expression. Conversely, if a matching failure occurs, then a $\mathsf{raise}\ \mathcal{T}_4$ expression is produced and the value $\mathcal{T}_4$ is propagated to the first declared try_with expression; at this point, if the handler $\mathcal{T}_2 \twoheadrightarrow \mathcal{T}_3$ is applicable to $\mathcal{T}_4$, then the exception is "captured", otherwise another $\mathsf{raise}\ \mathcal{T}_4$ will propagate the exception further towards the next try_with expression. As an example, evaluating the term below in a suitably modified natural semantics (here omitted) will yield the following judgment:

$$\mathsf{try}\ (\mathsf{try}\ (f(4,3) \twoheadrightarrow 5)\ f(3,4)\ \mathsf{with}\ f(4,X) \twoheadrightarrow X)\ \mathsf{with}\ f(3,X) \twoheadrightarrow X \Downarrow 4.$$

5 Conclusion

We have summarized in this paper the evolution of the rewriting calculus from its first version to the recent enhanced versions and discuss different extensions of the calculus.

We have shown how different classical strategies like, for example, innermost and outermost, can be defined using the different versions of the rewriting calculus. This has been done first by defining fix-point operators and adding a new operator, *first*, that selects a "successful" reduction. This operator resorts to be of foundational importance for expressing strategies and is also used in many languages expressing proof search tactics. Extending the calculus with an exception handling mechanism allows us to define the *first* operator in the calculus itself. The use of these operators provide us with a simple and precise operational semantics to the execution of rewrite rules and strategies of the language ELAN.

In fact, various approaches for handling the (matching) failure have been proposed for the different versions of the calculus and the representation of the *first* operator strongly depends on the chosen mechanism. However, the introduction of *first* in the calculus always seems to require some "external" feature: either we explicitly define this operator, or we encode it with an additional exception mechanism. One of our aims is to find a way of encoding *first* directly in the calculus, or at least have a good approximation of this operator without extending too much the initial frame.

We have shown that fix-point operators can be defined using the matching capabilities of ρCal and we have presented several examples of ad-hoc encodings of some TRS. We see this as a first step towards a simpler definition of *first* and an automatic encoding of TRS.

Moreover, we have seen that a suitable type system is able to type this fix-point operator as well as the various encodings we have shown in this paper, guaranteeing some "good behavior" of the considered ρ-terms without preventing the use of fix-point mechanisms.

References

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

[2] H. 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.

[3] P. Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France, October 1998. also TR LORIA 98-T-326.

[4] P. Borovansky, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, (285):155–185, July 2002.

[5] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. 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.

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

[7] 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.

[8] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[9] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, May 2001. Springer-Verlag.

[10] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In *Workshop on Rewriting Logic and its Applications - WRLA'2002, Pisa, Italy*. ENTCS, September 2002.

[11] H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. Submitted, 2003.

[12] 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.

[13] 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.

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

[15] G. Faure and C. Kirchner. Exceptions in the rewriting calculus. In S. Tison, editor, *Proceedings of the RTA conference*, volume 2368 of *Lecture Notes in Computer Science*, pages 66–82, Copenhagen, July 2002. Springer-Verlag.

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

[17] 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.

[18] G. Kahn. Natural Semantics. In *Proc. of STACS*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

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

[20] 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.

[21] L. Liquori. Semantica e Pragmatica di un Linguaggio Funzionale con le Continuazioni Esplicite. Laurea in Science dell'Informazione, University of Udine, 1990. In Italian, 74 pp.

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

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

[24] 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.

[25] 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.