ρ-Calculus. Its Syntax and Basic Properties

Horatiu Cirstea, Claude Kirchner

ρ-Calculus
Its Syntax and Basic Properties
Extended Abstract

Horatiu Cirstea Claude Kirchner

LORIA and INRIA
615, rue du Jardin Botanique, B.P. 101
54602 Villers-lès-Nancy Cedex, France
{Horatiu.Cirstea, Claude.Kirchner}@loria.fr

July 31, 1998

Abstract

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

1 Introduction

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

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

We come up with another point of view because of our previous works on the control of term rewriting [KKV95, Vit94, BKK98]. Indeed we realized that the tool that is needed in order to control rewriting should be made explicit and could be itself naturally described using rewriting. This leads to the concept presented in this paper which generalizes λ-calculus and rewriting and that we call ρ-calculus: the rewriting calculus.

In this calculus we can explicitly represent the application of a rule (say $a \to b$) to a term (like the constant a) as the object $[a \to b](a)$ which evaluates to the singleton $\{b\}$. But the application of a rule may fail like in $[a \to b](c)$ that evaluates to the empty set $\emptyset$. Of course variables may be used in rewrite rules like in $[f(x) \to x](f(a))$ that evaluates to $\{a\}$. In fact when evaluating this expression, the variable x is bound to a via a mechanism classically called matching and we have shown in [BKK98] how a rewrite rule can be considered as a function with matching as the parameter passing mechanism. The first simple but useful remark in this paper consists of noticing that a λ-expression $\lambda x.t$ could be represented as the rule $x \to t$. Indeed the β-reduction of a redex $(\lambda x.t\ u)$ is nothing more than $[x \to t](u)$ which reduces to $\{t\{x/u\}\}$. Of course we have to make clear what a substitution like $\{x/u\}$ is and how it applies to a term. But there is no surprise here and we consider a substitution mechanism that preserves the correct variable bindings via the appropriate α-conversion. In order to make this point clear in the paper, as in [DHK95], we will make a strong distinction between substitution (which takes care of variable

binding) and grafting (that performs replacement directly). So basic ρ-calculus objects are built from the signature, the abstraction operator $\rightarrow$ and the application operator [ ]( ). For example the well-known $(\omega\omega)$ λ-term is written in the syntax of ρ-calculus $[x \rightarrow [x](x)](x \rightarrow [x](x))$.

To summarize, in ρ-calculus, matching is used for binding variables to their actual values, abstraction is handled via the arrow binary operator and substitution takes care of variable bindings. It is thus a higher-order calculus like the λ-calculus and in apparent opposition to first-order rewriting. We will see later why this is not the case and why it allows us to understand the "variable separation" conditions traditionally imposed between first-order rules and the terms on which they are applied.

A ρ-calculus term (later called ρ-term) contains all the rewrite rules needed for its evaluation. This is usual for λ-calculus, this is generally not the case for term rewriting where the rewriting system is implicit. For example, assuming the existence of all the normal forms involved, the sentence "the β-normal form of a term t" is self-explanatory (except for the strategy used), as the sentence "the R-normal form of the term t" needs to make more precise the term rewrite system R and the strategy guiding the rewrite rules. In ρ-calculusthe rewrite rules and the strategy guiding them is coded in the ρ-term. For example, if we consider the rules $a \rightarrow b$ and $f(x) \rightarrow x$ the normal form of the ρ-term $[\mathsf{leftMostInnerMost}(\mathsf{first}(\mathsf{a} \rightarrow \mathsf{b}, \mathsf{f(x)} \rightarrow \mathsf{x}))](t)$ is the ρ-term $\{t'\}$. The term t' represents the normal form of the term t in the rewrite system $R = \{a \rightarrow b, f(x) \rightarrow x\}$ with a leftmost innermost strategy and strategy of trying to apply first the rule $\mathsf{a} \rightarrow \mathsf{b}$ and afterwards the rule $f(x) \rightarrow x$.

Another important feature of the ρ-calculus is its ability to handle non-determinism in the sense of sets of results. This is achieved via the explicit handling of reduction result sets, including the empty set that records failure of rule application. It allows us to make use in an explicit and direct way of non-terminating or non-confluent (equational) rewrite systems. For example if the symbol + is assumed to be commutative then applying the rule $x + y \rightarrow x$ to the term $a + b$ results in $\{a, b\}$ since there are two different ways to apply this rule modulo commutativity, and it gives two different results. This ability to integrate specific computations in the matching process allows us for example to use ρ-calculus for deduction modulo purposes like proposed in [DHK98].

The ρ-calculus is confluent when its evaluation rules are guided by an innermost strategy. This relies on the appropriate handling of non-determinism using sets of results. What is confluent is of course the calculus, not the rewrite systems it applies. So the ρ-normal form of the term a for the set of rules $\{a \rightarrow b, a \rightarrow c\}$ is $\{b, c\}$, which is unique. This is of course different from the R-normal forms of a which are b and c.

The purpose of this paper is to introduce the ρ-calculus, its syntax and deduction rules and to show how it can be used in order to naturally encode λ-calculus and standard, possibly conditional, term rewriting and finally why it provides an easy semantics for the rewrite based language ELAN.

In the first part we introduce the basic ρ-calculus and its syntax and deduction rules. We emphasize in particular here the fundamental role of the matching theory. We show in section 3 how ρ-calculus can be used to encode in a uniform way term rewriting and λ-calculus. Then in section 4 we enrich the basic calculus with primitives allowing us to describe more complex objects like choice operators. In the last part, we use the ρ-calculus to give an operational semantics to the ELAN language. This permits in particular to give a full account of the strategy objects and their use. We conclude by providing some of the research directions that are of main interest in the context of ELAN and more generally of rewrite based languages like ASF+SDF, ML, Maude or CafeOBJ.

The full version of this paper is available on the web[1].

2 The Basic ρ-calculus

A calculus is defined by four components:

- First its syntax that makes precise the formation of the terms manipulated by the calculus as well as the substitutions that are used during the evaluation mechanism. In the case of ρ-calculus, the core of the term formation relies on rewrite rules and rule application.
- The description of the substitution application on terms. This description is often given at the meta-level, except for explicit substitution frameworks like the one we describe in a later part of this paper. In the case of ρ-calculus, substitutions are higher-order substitutions and not grafting.

[1] `http://www.loria.fr/ cirstea/roCalculus1.ps.gz`

- The matching algorithm used to bind variables to their actual arguments. In the case of ρ-calculus, this is in general higher-order matching and in practical cases pattern and equational matching.
- The rules describing the way the calculus operates. It is the glue between the previous components and the simplicity and clarity of these rules are fundamental for the calculus usability.

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

2.1 Syntax

Definition 2.1 We consider a set $\mathcal{F} = \bigcup_m \mathcal{F}_m$ of ranked function symbols, where $\mathcal{F}_m$ is the subset of function symbols of arity m, and $\mathcal{X}$ a set of variables. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the set of first-order terms built on $\mathcal{F}$ using the variables in $\mathcal{X}$. The set of basic ρ-terms, denoted $\mathcal{R}(\mathcal{F}, \mathcal{X})$, is the smallest set of objects formed according to the following rules:

- $\mathcal{X}$ (variables) and $\mathcal{T}(\mathcal{F}, \mathcal{X})$ are ρ-terms,
- if $t_1, \ldots, t_n$ are ρ-terms and $f \in \mathcal{F}_m$ then $f(t_1, \ldots, t_m)$ is a ρ-term,
- if $t_1, \ldots, t_n$ are ρ-terms then $\{t_1, \ldots, t_n\}$ is a ρ-term (if $n = 0$ we have the ρ-term $\emptyset$),
- if t and u are ρ-terms then $[t](u)$ is a ρ-term (application of the ρ-term t to the ρ-term u),
- if t and u are ρ-terms then $t \to u$ is a ρ-term (rewrite rule).

The ρ-terms can also be inductively defined using a BNF syntax:

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

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

We adopt a very general discipline for the rule formation, and we do not enforce any of the standard restrictions like non-variable left-hand-sides or occurrence of the right-hand-side variables in the left-hand-side. We also allow rules containing rules as well as rule application.

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

Example 2.1 Some more examples of ρ-terms are:

- $[a \to b](a)$; this denotes the application of the rule $a \to b$ to the term a.
- $[f(x, y) \to g(x, y)](f(a, b))$; a classical rule application.
- $[x \to y](a)$; as expected, we will see later why the result is $\{y\}$.
- $[x \to (x \to x)]([x \to y](a))$; a more complicated ρ-term similar to the λ-term $((\lambda x.\lambda x.x)\ (\lambda x.y\ a))$.
- $[x \to [x](x)](x \to [x](x))$; the well-known $(\omega\omega)$ λ-term.
- $[(a \to b) \to c](a)$; a more complicated ρ-term without corresponding λ-term.

Definition 2.2 The set of free variables of a ρ-term t is denoted by $FV(t)$ and is defined by:

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

As in λ-calculus, α-conversion should be used in order to obtain a correct substitution calculus and the first order substitution (called here grafting) is not directly suitable for our calculus. We consider the usual notions of α-conversion and higher-order substitution as defined for example in [DHK95].

2.2 Matching

Computing the matching substitutions from a ρ-term t to a ρ-term t' is the fundamental parameter in the evaluation rules of the ρ-calculus.

For a theory T a *T-match-equation* is a formula of the form $t \ll^?_T t'$, where t and t' are ρ-terms. A substitution σ is a solution of the T-match-equation $t \ll^?_T t'$ if $T \models \sigma(t) = t'$. A *T-matching system* is a conjunction of T-match-equations. A substitution is solution of a T-matching system P if it is a solution of all the T-match-equations in P. We denote by $\mathbb{F}$ a T-matching system without solution.

Since in general we consider higher-order terms as well as arbitrary equational theories, T-matching is in general undecidable, even when restricted to first-order equational theories [JK91].

A T-matching system is called *trivial* when all substitutions are solution of it. We define the function *Solution* on a T-matching system $\mathcal{S}$ as returning the set of all T-matches of $\mathcal{S}$ when $\mathcal{S}$ is not trivial and $\{\mathbb{ID}\}$ where $\mathbb{ID}$ is the identity substitution when $\mathcal{S}$ is trivial. When the matching algorithm fails *Solution* returns the empty set ($\emptyset$).

One can think at using constraints as in constrained higher-order resolution [Hue73] or constrained deduction [KKR90] if one wants to use undecidable matching theories. But we are primarily interested here in the decidable cases. Among them we can mention higher-order pattern matching that is decidable and unitary as a consequence of pattern unification [Mil91, DHKP96], higher-order matching which is known to be decidable up to the fourth order [Pad96, Dow92, HL78] (the decidability of the general case being still open) , many first-order equational theories including associativity, commutativity, distributivity and most of their combinations [JK91].

Indeed we restrict ourselves later in this paper to just considering syntactic matching. In this case we assume that the left members of matching equations are composed only of first-order terms (i.e. not containing sets, arrows or applications). The syntactic matching substitution from t to t', when it exists, is unique and can be computed by a simple recursive algorithm given for example by G. Huet [Hue76]. It can also be computed by the following set of rules **SyntacticMatching** where the symbol $\wedge$ is assumed to be associative and commutative and where $\ll^?_T$ is denoted $\ll^?_\emptyset$.

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

SyntacticMatching: Rules for syntactic matching

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

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

Indeed this can be extended in a natural way when a symbol $+$ is commutative. In this case, the previous set of rules should be completed by:

$$\textbf{CommDec} \quad (t_1 + t_2) \ll^? (t'_1 + t'_2) \wedge P \quad \twoheadrightarrow \quad ((t_1 \ll^? t'_1 \wedge t_2 \ll^? t'_2) \vee (t_1 \ll^? t'_2 \wedge t_2 \ll^? t'_1)) \wedge P$$

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

$Fire$	$[l \to r](t)$	$\Longrightarrow$	$r\langle\langle Solution(l \ll^?_T t)\rangle\rangle$
$Distrib$	$[\{s_1, \ldots, s_n\}](t)$	$\Longrightarrow$	$\bigcup_{i=1}^n [s_i](t)$
$Batch$	$[s](\{t_1, \ldots, t_n\})$	$\Longrightarrow$	$\bigcup_{i=1}^n [s](t_i)$
$Switch_L$	$[\{l_1, \ldots, l_n\} \to r](t)$	$\Longrightarrow$	$\bigcup_{i=1}^n [l_i \to r](t)$
$Switch_R$	$[l \to \{r_1, \ldots, r_n\}](t)$	$\Longrightarrow$	$\bigcup_{i=1}^n [l \to r_i](t)$
Set	$f(A_1, \ldots, t, \ldots, A_n)$	$\Longrightarrow$	$\{f(a_1, \ldots, t, \ldots, a_n) \mid a_i \in A_i\}$
$Congruence$	$[f(s_1, \ldots, s_n)](f(t_1, \ldots, t_n))$	$\Longrightarrow$	$f([s_1](t_1), \ldots, [s_n](t_n))$
$Congruence\_fail$	$[f(s_1, \ldots, s_n)](g(t_1, \ldots, t_n))$	$\Longrightarrow$	$\emptyset$ if $f \neq g$
$Clash_D$	$[\emptyset](t)$	$\Longrightarrow$	$\emptyset$
$Clash_B$	$[t](\emptyset)$	$\Longrightarrow$	$\emptyset$
$Clash_{SL}$	$[\emptyset \to r](t)$	$\Longrightarrow$	$\emptyset$
$Clash_{SR}$	$[l \to \emptyset](t)$	$\Longrightarrow$	$\emptyset$
$Clash_S$	$f(\ldots, \emptyset, \ldots)$	$\Longrightarrow$	$\emptyset$

Figure 1: Basic ρ_T-calculus

$Propagate$	$r\langle\langle\{s_1, \ldots, s_n\}\rangle\rangle$	$\rightsquigarrow$	$\{s_1(r), \ldots, s_n(r)\}$
$Clash$	$r\langle\langle\emptyset\rangle\rangle$	$\rightsquigarrow$	$\emptyset$

Figure 2: Substitution rules

2.3 Deduction Rules

Given a rewrite rule $l \to r$, its application to a term t (denoted $[l \to r](t)$) consists of replacing the term t by σr, where σ is the substitution obtained by T-matching l on t and σr represents the application of the substitution on the term r as detailed above. The semantics of the application of a rewrite rule is given by the rules in Figure 1.

The $Fire$ rule *starts* the reduction process by T-matching the left-hand side of the rule on the argument of the rule. This generalizes the β-rule of λ-calculus. As we have seen before, the matching problem $l \ll^?_T t$ is solved in the theory T and depending on the theory T, we will obtain various rewriting calculi of interest.

The $Distrib, Batch, Switch$ rules and their $Clash$ versions describe the behavior of the application when one of the left-hand side, the right-hand side or the argument of the rule are (empty) sets.

The Set rule describes the semantics of a function with set arguments.

In order to push rule application deeper into terms, we introduce two $Congruence$ rules that deal with the application of a term of the form $f(t_1, \ldots, t_n)$ (where $f \in \mathcal{F}_n$) to another term of the same form.

The behavior of the "$\langle\langle\rangle\rangle$" operator is described by the rules in Figure 2. Note that these rules are applied at the meta-level and are not made explicit in the ρ-calculus. We are giving in a forthcoming paper a version of the ρ-calculus with explicit substitutions in the spirit of [ACCL91].

The *Propagate* rule yields as a result the union of the partial results obtained by applying each of the substitutions to the respective term. When the matching problem has no solution the rule *Clash* gives as result the empty set.

It should be pointed out that the interpretation of f is overloaded: in the left-hand side of the rule *Congruence* the first f operates on strategies and the second on terms; on the right-hand side the f operates on sets. Thus, we can say that for any symbol $f \in \mathcal{F}_n$ satisfying $f : A_1 \times \ldots \times A_n \to A$ we have two more symbols: $f_r : ((A_1 \to A_1) \times \ldots \times (A_n \to A_n)) \to (A \to A)$ which operates on strategies and $f_s : \{A_1\} \times \ldots \times \{A_n\} \to \{A\}$ which operates on sets.

Because it is already quite general and since we want to have a clear focus, in the rest of this paper we restrict to

- $T = \emptyset$ *i.e. to first-order syntactic matching,*
- *rewrite rules of the form* $l \to r$ *where* $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *i.e.* l *is a first-order term and thus does not contain any set, application or abstraction symbol.*

Notice that the last ρ-term presented in Example 2.1 does not satisfy this restriction.

Proposition 2.1 Under the above hypothesis, if the *Fire* rule is applied when none of the other rules is applicable and the left-hand side, right-hand side and argument of the rule are completely normalized then the ρ-calculus is confluent.

This restriction on the evaluation strategy is due to the handling of results sets and we believe that it can be weakened by explicitely describing the appropriate behavior of sets.

We conjecture that CRS and HRS [OR93] are specific ρ_T-calculi where the theory T consists of λ-patterns [Mil91].

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

We show in this section how the syntax and the deduction rules of the ρ-calculus can be restricted to some simpler calculi such as the λ-calculus and rewriting.

3.1 Encoding the λ-calculus

Let us consider the following restriction of the ρ-calculus, denoted ρ_λ-calculus:

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

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

Since the matching performed in this case in the rule *Fire* always succeeds, the rules *Clash* of the ρ-calculus are never applied.

The result of the matching from the rule *Fire* is always of the form $\{x \mapsto t\}$ so, the rules *Fire* and *Propagate* are equivalent to the simplified rule:

$$Fire' \quad [x \to r](t) \quad \Longrightarrow \quad \{\{x/t\}r\}$$

At this moment we can notice that the restricted ρ_λ-calculus and the λ-calculus are similar modulo the notations for the application, abstraction and substitution application. The rules *Fire'* and *Beta* both initiate the application of a substitution on a term; the other rules from the ρ_λ-calculus are auxiliary rules describing the behavior of sets.

The function φ that transforms terms in the syntax of the λ-calculus into the syntax of the ρ_λ-calculus is defined by:

$$\begin{array}{lcl} \varphi(\lambda x.t) & \mapsto & x \to t \\ \varphi(t\ u) & \mapsto & [t](u) \end{array}$$

Proposition 3.1 Given t and t' two λ-terms. Then, $t \longrightarrow_\beta t'$ iff $\varphi(t) \longrightarrow_\rho \{\varphi(t')\}$.

3.2 Encoding rewriting

We proceed similarly as for the λ-calculus and we restrict the syntax of the ρ-calculus to:

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

where $x \in \mathcal{X}$, $f \in \mathcal{F}$ and $l \in \mathcal{T}(\mathcal{F},\mathcal{X})$.

Proposition 3.2 Given t and t' two terms in $\mathcal{T}(\mathcal{F},\mathcal{X})$ and $\mathcal{R}$ a first order term rewrite system. If $t \xrightarrow{*}_{\mathcal{R}} t'$ then there exists a ρ-term u constructed using the rules in $\mathcal{R}$ such that $[u](t) \xrightarrow{*}_{\rho} \{t'\}$.

3.3 Encoding conditional rewriting

Conditional rewrite rules can be represented in an extended version of the ρ-calculus. The syntax is extended with a conditional operator **if** and with the boolean constants $True$ and $False$:

$\text{terms} \quad t \quad ::= \quad if \;\; t \mid True \mid False$

The following rules describe the semantics of the **if** operator:

$Ift \quad [if \;\; True](t) \quad \Longrightarrow \quad \{t\}$
$Iff \quad [if \;\; False](t) \quad \Longrightarrow \quad \emptyset$

For the first two rules we can give equivalent (implicit) definitions:

$Ift \quad if \;\; True \quad \Longrightarrow \quad id$
$Iff \quad if \;\; False \quad \Longrightarrow \quad fail$

The semantics of the conditions should be described by a set of rules. If we consider the equality operator the rules that should be defined are:

$Eq \quad a = a \quad \Longrightarrow \quad True$
$NEq \quad a = b \quad \Longrightarrow \quad False$, if $a \neq b$

A conditional rewrite rule of the form $(l \to r, \;\; if \;\; c)$ is represented by the ρ-term $(l \to [if \;\; c]r)$.

Proposition 3.3 Given t and t' two terms in $\mathcal{T}(\mathcal{F},\mathcal{X})$ and $\mathcal{R}$ a first order conditional term rewrite system. If $t \xrightarrow{*}_{\mathcal{R}} t'$ then there exists a ρ-term u constructed using the rules in $\mathcal{R}$ such that $[u](t) \xrightarrow{*}_{\rho} \{\ldots\{t'\}\ldots\}$.

4 The ρ-calculus

The basic ρ-calculus is already quite powerful because of the matching and substitution mechanisms it relies on, but we would like to enrich it further to allow an easy specification of choice operators. Indeed the whole calculus is intended to facilitate the specification and execution of non-deterministic possibly non-confluent rewrite systems as needed for concurrent systems as well as general deduction systems like theorem provers or constraint solvers.

The basic calculus could be enriched in two ways. The first one is to consider a fixpoint operator, the second one consists of an extention of the syntax and the inference rules of the calculus in order to integrate directly the desirable operators. In this paper we have chosen the second approach as it is more direct and close to what is done in ELAN as we will discuss later.

4.1 Primal strategies

Considering the ρ-terms as functions that apply rewrite rules, we called them *strategies* and since we are building an extention of the basic terms, we call *primal strategies* basic terms to which we add a sequence operator in order to denote the successive application of two strategies.

To this end, the syntax from section 2.1 is extended with the operators **id** and **fail** that correspond respectively to the identity rule $(x \to x)$ and the failure rule whose application fails on any term $(x \to \emptyset)$.

$$\text{terms} \quad t \quad ::= \quad id \mid fail$$

and the deduction rules are:

$$Identity \quad id \quad \Longrightarrow \quad x \to x$$

$$Fail \quad fail \quad \Longrightarrow \quad x \to \emptyset$$

The sequential application of two strategies is achieved by using the concatenation operator "**;**":

$$\text{terms} \quad t \quad ::= \quad t; t$$

The behavior of the concatenation operator is expressed by the following deduction rule:

$$Compose \quad [s_1; s_2](t) \quad \Longrightarrow \quad [s_2]([s_1](t))$$

Definition 4.1 *Primal strategies* are built starting from rewrite rules ($l \to r$) and the two operators id and $fail$ and using concatenation. Their application is defined by the rules in Figure 1, $Identity$, $Fail$ and $Compose$.

Example 4.1 If we consider the primal strategy

$$g(f(f(x) \to x)); g(f(x) \to x); g(a) \to a$$

then the only term on which the application of the strategy yields as result a non-empty set is $g(f(f(a)))$.

If the term is of the form $g(f(t))$ with t not of the form $f(t')$ then the $Congruence$ rule can be applied but the matching would fail and the result is the empty set. If t is of the form $f(t')$ with t' different from a then the application of the strategy $g(f(f(x) \to x)); g(f(x) \to x)$ to the term would yield as result $g(t')$ and we obtain once again the $\emptyset$ set due to the rule $Match\_Fail$. In the other cases the rule $Congruence\_fail$ would lead immediately to an empty set result.

4.2 Elementary strategies

The strategy language is now enriched by operators allowing us to describe the computation space in a convenient way.

The added operators select the final result(s) from a list of results and choose the strategy to be applied from a list of strategies. For the first type of selection we introduce the unary operators **one** and **all** that operate on strategies and whose semantics are described by the following rules:

$$\begin{array}{lllll} One & [one(s)](t) & \Longrightarrow & \{t'\} & \\ & & & \text{where } t' \in [s](t) & \\ All & [all(s)](t) & \Longrightarrow & [s](t) & \end{array}$$

Then we can also introduce selection operators **select-one**, **select-first**, **select-all** on strategy tuples ([BKK98]). Their semantics is defined by the rules:

$$\begin{array}{lllll} Select\_first & [select\_first(s_1, \ldots, s_n)](t) & \Longrightarrow & [s_j](t) & \\ & & & \text{if } \bigcup_{i=1}^{j-1}[s_i](t) = \emptyset \text{ and } [s_j](t) \neq \emptyset & \\ Select\_one & [select\_one(s_1, \ldots, s_n)](t) & \Longrightarrow & [s_i](t) & \\ & & & \text{if } [s_i](t) \neq \emptyset & \\ Select\_all & [select\_all(s_1, \ldots, s_n)](t) & \Longrightarrow & \bigcup_{i=1}^{n}[s_i](t) & \end{array}$$

The operator **select-first** returns the result of the first successful application, **select-one** selects one of the successful applications and **select-all** provides the union of all (successful) applications. These selection operators allow us to describe the operators **first**, **dc**(dont care) and **dk**(dont know) by the rules:

$$Dk \quad [dk(s_1,\ldots,s_n)](t) \Longrightarrow [select\_all(all(s_1),\ldots,all(s_n))](t)$$

$$Dc \quad [dc(s_1,\ldots,s_n)](t) \Longrightarrow [select\_one(all(s_1),\ldots,all(s_n))](t)$$

$$First \quad [first(s_1,\ldots,s_n)](t) \Longrightarrow [select\_first(all(s_1),\ldots,all(s_n))](t)$$

Intuitively, the selectors are used to select the strategy and the **one** and **all** operators determine the number of results to be returned when applying the respective strategy to the input term.

Some other strategies can be constructed starting from the above operators. For example, an elementary strategy that chooses one result in each set of results provided by a sequence of strategies would cut dramatically the number of reductions is :

$$One \quad [one(s_1,\ldots,s_n)](t) \Longrightarrow [select\_one(one(s_1),\ldots,one(s_n))](t)$$

Note also that the result selector can be made more precise when a different representation for results is chosen. For example, when using lists, the **one** operator could be completed with "nth" ones.

4.3 Defined strategies

Using this kind of operators and primal strategies we can define more elaborate strategies. For example, a function **map** can be defined in an explicit way by:

$$Map_1 \quad [map(s)](nil) \Longrightarrow nil$$
$$Map_2 \quad [map(s)](a.l) \Longrightarrow [s](a).[map(s)](l)$$

or in an implicit (and more compact) way:

$$Map \quad map(s) \Longrightarrow first(nil,\ s.map(s))$$

Other basic strategy definitions like **iterate** or **repeat** can be given in a similar way.

More complicated control structures can be constructed if we extend the $Congruence$ with new cases that deal with constant head symbols in the strategy application:

$$Congruence\_M \quad [\Phi(s_1,\ldots,s_n)](f(t_1,\ldots,t_n)) \Longrightarrow f([s_1](t_1),\ldots,[s_n](t_n))$$

$$Congruence\_S \quad [\Psi(s)](f(t_1,\ldots,t_n)) \Longrightarrow f([s](t_1),\ldots,[s](t_n))$$

where Φ, Ψ are constants of the language ($\Phi, \Psi \notin \mathcal{F}$) acting as wildcards for the head symbol of a strategy application.

A strategy operator applying a strategy in an *innermost* way is easy to define at this moment:

$$Im \quad im(s) \Longrightarrow \Psi(im(s)); s$$

while the *outermost* strategy has a similar description:

$$Om \quad om(s) \Longrightarrow s; \Psi(om(s))$$

Depending on the order used in evaluating the arguments of the function in the right-hand side of the rule $Congruence\_S$ the strategy im is a *leftmost innermost, rightmost innermost* or *random innermost* strategy.

5 A ρ-calculus semantics for the ELAN language

ELAN is an environment for specifying and prototyping deduction systems in a language based on labeled conditional rewrite rules, and strategies to control rule application. The ELAN system offers a compiler and an interpreter of the language. The ELAN language allows us to describe in a natural and elegant

way various deduction systems and to experiment with for example the combination of theorem provers, constraint solvers and decision procedures [Vit94, KKV95, BKK+96]. It has been experimented on several non-trivial applications ranging from constraint solvers to logic programming and automated theorem proving[2].

ELAN provides a kernel that implements the leftmost innermost standard rewriting strategy, the elementary strategies, and which allows the iteration of the construction on defined strategies as defined in the previous section.

A first semantics could be given to an ELAN program using rewriting logic [Mes92], but more conveniently ELAN's rules can be expressed using the ρ-calculus and thus an ELAN program is just a set of ρ-terms. A rule with no conditions and no local assignments $l \Rightarrow r$ is represented by $l \rightarrow r$. The local assignments are let-like constructions that allow applications of strategies on some terms. A typical ELAN rule is of the form:

[label] $l \Rightarrow r$
if condition
where x:= [strategy] term

For example the ELAN rewrite rule:

[deriveSum] $p_1 + p_2 \Rightarrow p_1' + p_2'$
where $p_1' := [derive]p_1$
where $p_2' := [derive]p_2$

is represented by the ρ-term $p_1 + p_2 \Rightarrow [derive]p_1 + [derive]p_2$.

As detailed in the full version of the paper, the semantics of the primal and elementary strategies of ELAN are reflected by the corresponding strategies in the ρ-calculus and the user defined strategies are similar to the ρ defined strategies.

6 Conclusion

The ρ-calculus introduced in this paper is a calculus of rule application. It allows the rule syntax to be very general, allowing λ-calculus as well as first-order rewriting to be simply embedded in it.

In this paper, the ρ-calculus syntax and elementary properties have been presented in the untyped case. We are currently extending these results to the typed case and we are also building an explicit substitution version of the calculus in the spirit of λ-calculus with explicit substitutions.

We believe that this calculus is both conceptually simple as well as very powerful. This allowed us to show how the ρ-calculus can be used as the operational semantics of ELAN. More generally, the applications to many other frameworks have to be investigated, including rewrite based languages like ASF+SDF, ML, Maude or CafeOBJ but also non-deterministic transition systems.

Among the many research directions concerned by the use of the ρ-calculus for the combination of first-order and higher-order paradigms we are now deepening the relationship of ρ-calculus with rewriting logic [Mes92] and we plan to investigate the relationship of this calculus with higher-order rewrite concepts like CRS and HOR as mentioned earlier.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

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

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

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

[2] see `http://www.loria.fr/equipes/protheo/PROJECTS/ELAN/elan.html` for more details

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[OR93] V. v. Oostrom and F. f. Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In *HOA'93*, volume 816 of *Lecture Notes in Computer Science*, pages 276–304. Springer-Verlag, 1993.

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

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

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