Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 117 (2005) 51–67

www.elsevier.com/locate/entcs

A ρ-Calculus of Explicit Constraint Application

Horatiu Cirstea[b], Germain Faure[a] and Claude Kirchner[b]

[a] *LORIA, ENS-Cachan Campus Ker Lann, F-35170 Bruz, France* Germain.Faure@loria.fr

[b] *LORIA & NANCY II & INRIA, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex France* {Horatiu.Cirstea, Claude.Kirchner}@loria.fr

Abstract

Theoretical presentations of the ρ-calculus often treat the matching constraint computations as an atomic operation although matching constraints are explicitly expressed. Actual implementations have to take a much more realistic view: computations needed in order to find the solutions of a matching equation can be really important in some matching theories and the substitution application usually involves a term traversal.
Following the works on explicit substitutions in the λ-calculus, we propose, study and exemplify a ρ-calculus with *explicit constraint handling*, up to the level of substitution applications. The approach is general, allowing the extension to various matching theories. We show that the calculus is powerful enough to deal with errors. We establish the confluence of the calculus and the termination of the explicit constraint handling and application sub-calculus.

Keywords: ρ-calculus, matching, constraints, explicit substitutions.

Introduction

Pattern matching occurs in many programming languages as a powerful tool to express requirements about the arguments of a program (*e.g.* ELAN [4], Maude [5], TOM [20], ML) and the computational behavior of a calculus can be really influenced by its ability to perform pattern matching [9]. Many works studied the matching but most of the time they do not use it in a full way, *i.e.*, they do not explicitly express and therefore do not exploit matching failures. This ability to express matching failures is a key point in the ρ-calculus.

The ρ-calculus was introduced to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule (*abstraction*), rule *application*

doi:10.1016/j.entcs.2004.06.029

and *result*. In the ρ-calculus, the usual λ-abstraction $\lambda X.B$ is replaced by a rule abstraction $A \twoheadrightarrow B$, where A is an arbitrary term and B is the argument to be fired, and the free variables of A are bound in B.

The matching power of the ρ-calculus can be regulated using arbitrary theories. In classical term rewriting, this can lead to non-deterministic behavior but since "results" are first class citizens in the ρ-calculus, we can represent all possible results as a single one (using the structure operator denoted by ";"). The way these results are represented is also a parameter of the calculus since different semantics are obtained according to the theories associated to the structure operator. Typically, if an associative-commutative and idempotent status is given to this operator then, we recover the semantics of result *sets* [7]. If one prefers lists or multisets, then the corresponding formalization should be specified.

Matching failures can be treated in different ways. For example, every matching failure can be reduced to a special term representing *the failure* term. This may cause problems *w.r.t.* the calculus confluence and this is why a major evolution in the syntax and capability of the calculus is proposed in [8]: *delayed matching constraints* become an explicit part of the calculus. So, the application of an abstraction $A \twoheadrightarrow B$ to a term C, denoted $(A \twoheadrightarrow B)C$, evaluates to $(A \ll C)B$ which represents a term where the matching constraint $A \ll C$ is "put on the stack" for later evaluation and use. When at least a solution exists, the delayed matching constraint is solved at the meta-level and then the delayed matching constraint is evaluated to $\sigma_1(B); \ldots; \sigma_n(B) \ldots$ where σ_i, $i = 1 \ldots n$, are the solutions of the matching between A and C. If no solution exists, the delayed matching constraint is not reduced.

The ability to parameterize the ρ-calculus by a matching theory opens new possibilities and leads to a very expressive calculus. Nevertheless, it is surprising that all the computations related to the considered matching theory still belong to the meta-level. The same situation arises in Rogue [24], a new programming language based on an untyped version of the ρ-calculus and primary intended for implementing decision procedures. The operational semantics of Rogue as well as the rules of the ρ-calculus use helper functions that are indeed implicit computations. These computations are conceptually and computationally important in all matching theories, from syntactic ones to quite elaborated ones like associative-commutative theories [12]. Therefore, to make explicit the handling of constraints, one must make explicit the matching and the constraint (substitution) application. This leads to a calculus *à la* $\lambda_{\mathtt{x}}$-calculus [23] simple and without substitution compositions. We call it the $\rho_{\mathtt{x}}$-calculus.

The λ-calculus with explicit substitutions has been widely studied and

provides a nice tool to deal with higher order unification [11] or to represent incomplete proofs in type theory [21]. As far as implementation issues are concerned, explicit substitution calculi are very important [18].

In all the explicit substitution calculi [1,17,23], substitutions can be delayed thanks to the *Beta* rule that transforms a β-redex $(\lambda x.a)b$ into the *explicit* application on a of the substitution that replaces x by b. In the ρ-calculus, matching constraints can be delayed too, thanks to a rule that does not compute anything but transforms the application of a rewrite rule into the application of a matching constraint. It therefore makes explicit the decision to reduce a given redex and it provides the capacity to decide when one wants to start the computations needed to apply a rewrite rule.

Then, the matching constraints are computed and applied in one step. In concrete implementations these operations should be separated and should interact with other computations and, in particular, we want computations on constraints and applications of constraints to be explicit. We can think of an implementation of the ρ-calculus with explicit computations and applications of constraints as using a scheduler that switches regularly between computations on constraints, applications of substitutions and the basic evaluation rules.

Contributions: We propose an extension of the ρ-calculus to deal explicitly with matching constraints. This calculus enjoys the usual good properties of explicit substitutions (conservativity, termination) and it is confluent. We show that the ρ-calculus, and especially the $\rho_{\mathtt{x}}$-calculus, are suitable as a useful theoretical back-end for implementations.

Road-Map: The first two sections describe respectively the syntax and the semantics of the $\rho_{\mathtt{x}}$-calculus presenting its motivation and construction. In Section 3 we give examples of the behavior of the calculus focusing mainly on the handling of errors. The next section presents the main properties of the calculus such as the confluence of the calculus for linear patterns. In Section 5 we discuss some extensions of the calculus introduced in order to increase its expressiveness and efficiency. We conclude by presenting related and future works.

1 Syntax of the $\rho_{\mathtt{x}}$-calculus

The syntax of the $\rho_{\mathtt{x}}$-calculus presented in Figure 1 is an extension of the one used for the plain ρ-calculus where the left-hand side of an *abstraction* (built using the "$\twoheadrightarrow$" operator) defines the variables we abstract on and some context

information and where terms can be grouped together into *structures* (built using the operator ";"). A term in a left-hand side of an abstraction is often called a *pattern*. Several new constructions are added to the original syntax:

- Constraints become first-class objects of the calculus and are conjunctions (built with the operator "$\wedge$") of matching problems of the form $A \ll B$. This way matching problems can be explicitly decomposed.
- The application operator denoted by concatenation is extended to constraint application.
- We use a special symbol "$\{\}$" to denote the application of substitutions on terms and constraints. Using this syntax, explicit treatment (propagation) of substitutions can be done.

We assume that the application operator associates to the left, while the other operators associate to the right. The priority of the application is higher than that of "$\twoheadrightarrow$" which is higher than that of ";" which is of higher priority than the "$\{\}$" which is, in turn, of higher priority than the "$\wedge$". The symbols $A, B, C, \ldots$ range over the set $\mathcal{T}$ of terms, the symbols $X, Y, Z, \ldots$ range over the set $\mathcal{V}$ of variables ($\mathcal{V} \subseteq \mathcal{T}$), the symbols $a, b, c, \ldots, f, g, h$ range over a set $\mathcal{K}$ of constants ($\mathcal{K} \subseteq \mathcal{T}$). We call *algebraic* the terms of the form $(\ldots((f\ A_1)\ A_2)\ldots)\ A_n$ with $f \in \mathcal{K}$ and we usually denote them by $f(A_1, A_2, \ldots, A_n)$. A *domain* of a match-equation $A \ll B$ (resp. of a conjonction of constraints) is the set of free variables of A (resp. the union of the domains of each match-equation of the conjonction). We denote it by $\mathcal{D}om(A \ll B)$.

As in any calculus involving binders, we work modulo the α-*conversion* of Church, and modulo the *hygiene-convention* of Barendregt [2], *i.e.*, free and bound variables have different names.

To support the intuition, one can mention that the application of a rewrite rule (abstraction) to a term is always evaluated to the application of the corresponding constraint to the right-hand side of the rewrite rule. Such a construction was called a *delayed matching constraint* in the classical ρ-calculus. A constraint application will be transformed, if possible, into a collection of substitution applications.

For the sake of simplicity, in this article, substitutions are constraints of the form $X \ll A$. Of course, the set of substitutions can be easily extended to more sophisticated constraints like, for example, $X \ll A \wedge Y \ll B$. The main benefit of this latter approach is that several substitution applications implying several term traversals can be merged together leading thus to more efficient implementations [18].

Example 1.1 (Encoding of λ-terms)

Terms	A, B	$::= \mathcal{V}$	(Variables)
		$\mid \mathcal{K}$	(Constants)
		$\mid A \twoheadrightarrow B$	(Abstraction)
		$\mid A\ B$	(Functional application)
		$\mid (\mathcal{C})B$	(Constraint application)
		$\mid A; B$	(Structure)
		$\mid \{X \ll A\}B$	(Substitution application on terms)
Constraints	$\mathcal{C}, \mathcal{D}$	$::= A \ll B$	(Match-equation)
		$\mid \mathcal{C} \wedge \mathcal{D}$	(Conjunction of constraints)
		$\mid \{X \ll A\}\mathcal{C}$	(Substitution application on constraints)

where $\wedge$ is supposed to be associative-commutative and idempotent.

Figure 1. Syntax of ρ_x-calculus

One can encode the λ-calculus in the ρ-calculus. The binder "λ" is replaced by a rule abstraction "$\twoheadrightarrow$".

λ-calculus	ρ-calculus
$\lambda X.X$	$X \twoheadrightarrow X$
$\lambda X.\lambda Y.X$	$X \twoheadrightarrow Y \twoheadrightarrow X$
$\lambda X.(XX)$	$X \twoheadrightarrow XX$

Example 1.2 (Encoding of first-order terms)
Using the constants $\mathtt{tt}, \mathtt{ff}, \mathtt{not}, \mathtt{and}, \mathtt{or}$ (denoting respectively the boolean values true and false, the negation, the conjunction and the disjunction) we can define the following first-order terms: $\mathtt{and}(X,\mathtt{tt})$ and $\mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))$.

Example 1.3 (Rewrite rules)
Some rules to compute in the Boolean algebra:

- $\mathtt{and}(X,\mathtt{tt}) \twoheadrightarrow X$; the free variable of the pattern $\mathtt{and}(X,\mathtt{tt})$ is bound in the body X of the abstraction.
- $\mathtt{not}(\mathtt{and}(X,Y)) \twoheadrightarrow \mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))$; this rule bounds the variables X and Y.
- $\mathtt{xor}(X,X) \twoheadrightarrow \mathtt{ff}$; a non-linear rule.

Example 1.4 (Constraint application)
The application of the second rewrite rule given in Example 1.3 to the term $\mathtt{not}(\mathtt{and}(\mathtt{tt},\mathtt{ff}))$ is denoted as $\big(\mathtt{not}(\mathtt{and}(X,Y)) \twoheadrightarrow \mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))\big)\ \mathtt{not}(\mathtt{and}(\mathtt{tt},\mathtt{ff}))$. We will see in the next sections that this term can be successively reduced

to the constraint application $\big(X \ll \mathtt{tt} \wedge Y \ll \mathtt{ff}\big)\,\mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))$ and afterwards, to the substitution applications $\{X \ll \mathtt{tt}\}\big(\{Y \ll \mathtt{ff}\}\mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))\big)$ and finally, to the term $\mathtt{or}(\mathtt{not}(\mathtt{tt}),\mathtt{not}(\mathtt{ff}))$.

2 Semantics of the $\rho_{\mathtt{x}}$-calculus

In the classical ρ-calculus, when reducing the application of a constraint to a term, *i.e.*, the delayed matching constraint, the corresponding matching problem is solved and the resulted substitutions are applied at the meta-level of the calculus. This means that, in one step, we compute the substitution from the matching constraint and apply it.

This reduction can be obviously decomposed into two steps, one computing the substitution and the other one describing the application of the corresponding substitution. This decomposition does not mean that the matching computations leading from constraints to substitutions and the application of the substitution are explicit but just that they are clearly separated. Depending on the matching theory, these computations can be really significant and we want to go further on and to make them explicit.

The small-step reduction semantics of the $\rho_{\mathtt{x}}$-calculus is given in Figure 2 where the reduction rules of the calculus are split into three categories:

- Rules describing the application of structures and abstractions on ρ-terms.
- Rules that describe the solving of the matching problems (*i.e.* their reduction to a normal form) and that trigger the application of the resulted substitutions (constraints).
- Rules defining the application of substitutions.

Term application

The two first rules (ρ) and (δ) are inherited from the plain ρ-calculus. The rule (δ) deals with the distributivity of the application on the structures built with the ";" operator (see Example 3.2) while the rule (ρ) reduces the application of an abstraction to a term to the right-hand side of the rewrite rule constrained by a matching problem.

Constraint computation

The *Decomposition* set of rules is strongly related to the considered matching theory which is a parameter of the ρ-calculus. Since there exists no generic algorithm to decide/solve matching constraints, we have to make precise here the considered theory (or theories). For simplicity, we have chosen to present the $\rho_{\mathtt{x}}$-calculus with an empty theory and thus we introduce the rules

Term application

(ρ) $(A \twoheadrightarrow B)\ C \to (A \ll C)\ B$

(δ) $(A; B)\ C \to A\ C; B\ C$

Constraint computation

Decomposition

$(Decompose_{;})$ $A_1; A_2 \ll B_1; B_2 \to A_1 \ll B_1 \wedge A_2 \ll B_2$

$(Decompose_{\mathcal{K}})$ $f(A_1, \ldots, A_n) \ll f(B_1, \ldots, B_n) \to A_1 \ll B_1 \wedge \ldots \wedge A_n \ll B_n$

Constraint application

$(ToSubst_{\wedge})$ $(X \ll A \wedge \mathcal{C})B \to (\mathcal{C})(\{X \ll A\}B)$ if $X \notin \mathcal{D}om(\mathcal{C})$

$(ToSubst_{\ll})$ $(X \ll A)B \to \{X \ll A\}B$

$(ToSubst_{Id})$ $(a \ll a)B \to B$

Substitution application

$(Replace)$ $\{X \ll A\}X \to A$

$(Eliminate_{\mathcal{V}})$ $\{X \ll A\}Y \to Y$ if $X \neq Y$

$(Eliminate_{\mathcal{K}})$ $\{X \ll A\}f \to f$

$(Share\ )$ $\{X \ll A\}(B\ C) \to (\{X \ll A\}B)\ (\{X \ll A\}C)$

$(Share_{;})$ $\{X \ll A\}(B; C) \to \{X \ll A\}B; \{X \ll A\}C$

$(Share_{\twoheadrightarrow})$ $\{X \ll A\}(B \twoheadrightarrow C) \to B \twoheadrightarrow \{X \ll A\}C$

$(Share_{\ll})$ $\{X \ll A\}(B \ll C) \to B \ll \{X \ll A\}C$

$(Share_{\wedge})$ $\{X \ll A\}(\mathcal{C} \wedge \mathcal{D}) \to \{X \ll A\}\mathcal{C} \wedge \{X \ll A\}\mathcal{D}$

σ, κ, $\rho_{\mathtt{x}}$

Figure 2. Small-step reduction semantics of the $\rho_{\mathtt{x}}$-calculus

$(Decompose_{;})$ and $(Decompose_{\mathcal{K}})$ inspired by the well-known algorithms solving matching problems.

In the ρ-calculus there are higher-order symbols (*e.g.* "$\twoheadrightarrow$", "$\ll$") and we cannot do a first-order matching for this kind of symbols (or we should do it very strictly and for specific purpose). So, we will only decompose *algebraic terms* (*i.e.* applications with some constant head symbol) and structures.

As we mentioned before, the ρ-calculus is well-suited to deal with errors, represented by constraints without solution. This means that there exist con-

straints that do not represent substitutions [1]. Depending on the intended use of the calculus we may want or not to propagate (constraint) failures. If one propagates constraints without any solution, we would lose the error's location and we would obtain *in fine* a term with constraints without any solution applied on each leaf of the term (considered as a tree). The information contained in such a term seems useless to analyze the error and, for debugging reasons, we do not want to lose the error's location. This is why we need to identify the successful constraints whose applications represent (*i.e.* can be reduced to) substitution applications.

Once a certain "successful" normal form for a matching problem is obtained, the application of the corresponding substitution is triggered by the set of rules *Constraint application*.

To go from constraints to substitutions, we simplify a constraint (using decomposition rules) until we find a subpart of the constraint of the form $X \ll A$. If this non-decomposable constraint is independent of the remaining part of the constraint, we can "push it out" of the constraint and trigger its application as a substitution.

One can notice that the constraint $X \ll A \wedge Y \ll B$ is decomposed into two substitutions: $X \ll A$ and $Y \ll B$ and thus, in order to apply such a constraint to a term, we will need to visit the whole term twice.

One question still remains: how to deal with non-linearity? There are many answers but the simplest one is: wait until the problem becomes linear. Surprising as it may seem, this is the best way to deal with this difficulty. To be more precise, when we find a constraint of the form $X \ll A \wedge X \ll B$, since none of the rules ($ToSubst$) can be applied ($X \in \mathcal{D}om(X \ll B)$) we try to reduce A and/or B until we obtain two equal terms. Therefore, we eventually obtain the constraint $X \ll A'$ (since $\wedge$ is assumed to be idempotent) or a matching constraint that cannot be applied (see Example 3.3).

Substitution application

The application of a substitution is defined by straightforward rules that distribute this application over the different operators of the calculus and replace accordingly the concerned variables. Since we work modulo α-conversion, when we apply a substitution to an abstraction, the left-hand side of the abstraction is not affected. Similarly, the left-hand sides of matching equations (normally issued from abstraction applications) are not concerned by the substitution application.

[1] Of course, in the λ-calculus, such questions do not arise since we only consider trivial matching problems that are always successful and, to propagate the corresponding substitution (*i.e.*, to apply it step by step) always makes sense.

We should point out that the constraint application operator cannot be overloaded to handle substitutions as well and that a special symbol to denote the substitution application is needed. Otherwise, a terminating and confluent rule system for the explicit application of substitutions seems difficult to achieve.

As usual, we define the one step $\mapsto_{\mathcal{R}}$ and many steps $\mapsto\!\!\!\!\to_{\mathcal{R}}$ relations *w.r.t.* a set of rules $\rightarrow_{\mathcal{R}}$. This way we define the relations $\mapsto_{\sigma}, \mapsto_{\kappa}$ and $\mapsto_{\rho_{\mathtt{x}}}$ induced by the rules in Figure 2.

3 Examples

In this section, we will give examples illustrating the behavior of our calculus. In [13], it is shown that the $\rho_{\mathtt{x}}$-calculus embeds the λ_x-calculus. The next example illustrates the atomicity of the application of a rewrite rule.

Example 3.1 (Application of a rewrite rule)
In order to compute the disjunctive normal form, we use the rewrite rule $\mathtt{not}(\mathtt{and}(X,Y)) \twoheadrightarrow \mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))$. The application of this rewrite rule to the term $\mathtt{not}(\mathtt{and}(\mathtt{tt},\mathtt{ff}))$ is described in the $\rho_{\mathtt{x}}$-calculus by the following reduction:

$$
\begin{array}{lll}
 & \big(\mathtt{not}(\mathtt{and}(X,Y)) \twoheadrightarrow \mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))\big)\ \mathtt{not}(\mathtt{and}(\mathtt{tt},\mathtt{ff})) & \\
\mapsto_{\rho} & \big(\mathtt{not}(\mathtt{and}(X,Y)) \ll \mathtt{not}(\mathtt{and}(\mathtt{tt},\mathtt{ff}))\big)\ \mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y)) & \\
\mapsto\!\!\!\!\to_{Decompose_{\mathcal{K}}} & \big(X \ll \mathtt{tt} \wedge Y \ll \mathtt{ff}\big)\ \mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y)) & \\
\mapsto_{ToSubst_{\wedge}} & (X \ll \mathtt{tt})\big(\{Y \ll \mathtt{ff}\}\mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))\big) & \\
\mapsto_{ToSubst_{\ll}} & \{X \ll \mathtt{tt}\}\big(\{Y \ll \mathtt{ff}\}\mathtt{or}(\mathtt{not}(X),\mathtt{not}(Y))\big) & \\
\mapsto\!\!\!\!\to & \{X \ll \mathtt{tt}\}\big(\mathtt{or}(\mathtt{not}(X),\mathtt{not}(\mathtt{ff}))\big) & \textit{Subst. app.} \\
\mapsto\!\!\!\!\to & \mathtt{or}(\mathtt{not}(\mathtt{tt}),\mathtt{not}(\mathtt{ff})) & \textit{Subst. app.}
\end{array}
$$

Example 3.2 (Application of a rewrite system)
We show how a structure of rewrite rules applies to a term. In a first approximation, this can be seen as the application of a rewrite system.

$$
\Big(\mathtt{and}(X,\mathtt{or}(Y,Z)) \twoheadrightarrow \mathtt{or}(\mathtt{and}(X,Y),\mathtt{and}(X,Z));
$$
$$
\mathtt{and}(\mathtt{or}(X,Y),Z) \twoheadrightarrow \mathtt{or}(\mathtt{and}(X,Z),\mathtt{and}(Y,Z))\Big)\ \mathtt{and}(\mathtt{or}(\mathtt{tt},\mathtt{ff}),\mathtt{or}(\mathtt{ff},\mathtt{ff}))
$$

We use the (δ) rule to distribute the two rewrite rules:

$$
\mapsto_{\delta} \big(\mathtt{and}(X,\mathtt{or}(Y,Z)) \twoheadrightarrow \mathtt{or}(\mathtt{and}(X,Y),\mathtt{and}(X,Z))\big)\ \mathtt{and}(\mathtt{or}(\mathtt{tt},\mathtt{ff}),\mathtt{or}(\mathtt{ff},\mathtt{ff}));
$$
$$
\big(\mathtt{and}(\mathtt{or}(X,Y),Z) \twoheadrightarrow \mathtt{or}(\mathtt{and}(X,Z),\mathtt{and}(Y,Z))\big)\ \mathtt{and}(\mathtt{or}(\mathtt{tt},\mathtt{ff}),\mathtt{or}(\mathtt{ff},\mathtt{ff}))
$$

and we finally obtain (by reducing each rule application as in Example 3.1):

$$\mathtt{or}(\mathtt{and}(\mathtt{or}(\mathtt{tt},\mathtt{ff}),\mathtt{ff}),\mathtt{and}(\mathtt{or}(\mathtt{tt},\mathtt{ff}),\mathtt{ff}))\ ;\ \mathtt{or}(\mathtt{and}(\mathtt{tt},\mathtt{or}(\mathtt{ff},\mathtt{ff})),\mathtt{and}(\mathtt{ff},\mathtt{or}(\mathtt{ff},\mathtt{ff})))$$

The application of a rewrite system is actually never as simple as presented above. Here, we encode only one (meta) rewriting step but in general the encoding is more complicated because one needs to encode the evaluation strategy. The problem is solved by using (typed) fixpoints to apply the rewrite system recursively (see [9] for a full presentation).

Example 3.3 (Application of a non-linear rewrite rule)
There are no restrictions related to the set of patterns that can be non-linear:

$$\begin{aligned}
& (\mathtt{xor}(X,X) \twoheadrightarrow \mathtt{ff})\ \mathtt{xor}(\mathtt{tt},\mathtt{tt}) \\
\mapsto_{\rho} \quad & \big(\mathtt{xor}(X,X) \ll \mathtt{xor}(\mathtt{tt},\mathtt{tt})\big)\ \mathtt{ff} \\
\mapsto_{Decompose_{\mathcal{K}}} \quad & \big(X \ll \mathtt{tt} \wedge X \ll \mathtt{tt}\big)\ \mathtt{ff} \equiv \big(X \ll \mathtt{tt}\big)\ \mathtt{ff} \qquad (\wedge \text{ is idempotent}) \\
\mapsto_{ToSubst_{\ll}} \quad & \{X \ll \mathtt{tt}\}\mathtt{ff} \\
\mapsto_{Eliminate_{\mathcal{K}}} \quad & \mathtt{ff}
\end{aligned}$$

Of course, the application of a non-linear rewrite rule may cause failures due to merging clashes. Merging clashes are not reduced and are kept as a constraint application failure.

Example 3.4 (Application of a non-linear rewrite rule)

$$\begin{aligned}
& (\mathtt{xor}(X,X) \twoheadrightarrow \mathtt{ff})\ \mathtt{xor}(\mathtt{tt},\mathtt{ff}) \\
\mapsto_{\rho} \quad & \big(\mathtt{xor}(X,X) \ll \mathtt{xor}(\mathtt{tt},\mathtt{ff})\big)\ \mathtt{ff} \\
\mapsto_{Decompose_{\mathcal{K}}} \quad & \big(X \ll \mathtt{tt} \wedge X \ll \mathtt{ff}\big)\ \mathtt{ff}
\end{aligned}$$

In the following examples, we will describe how a data structure can be defined and used in the ρ-calculus. We focus on the matching failure cases. We deal with the example of lists and to support the intuition, we give also the examples in ML syntax.

Example 3.5 (Destructors in O'CAML)
In O'CAML [25], the list destructors can be naturally written, using pattern-matching, as:

```
# let car l = match l with
    |x::m -> x;;
# let cdr l = match l with
    | x::m -> m;;
```

These are partial functions since we cannot apply them to `[]` without raising an exception which encodes the matching failure.

```
# car [];;
Exception: Match_failure ("", 12, 42).
```

Example 3.6 (Destructors in the ρ-calculus)
In the ρ-calculus, the data structure constructors are defined using constants. We will use the constants "Empty" and "Cons" to denote the list constructors corresponding in O'CAML to the use of "[]" and "::". In the ρ-calculus, the destructors are written: $\mathfrak{car} \triangleq \mathtt{Cons}(X, M) \twoheadrightarrow X$ and $\mathfrak{cdr} \triangleq \mathtt{Cons}(X, M) \twoheadrightarrow M$. The application of $\mathfrak{car}$ to the list $\mathtt{Cons}(\mathtt{a}, \mathtt{Empty})$ reduces to the constant a:

$$
\begin{array}{ll}
 & (\mathtt{Cons}(X, M) \twoheadrightarrow X)\ \mathtt{Cons}(\mathtt{a}, \mathtt{Empty}) \\
\mapsto_{\rho} & (\mathtt{Cons}(X, M) \ll \mathtt{Cons}(\mathtt{a}, \mathtt{Empty}))\ X \\
\mapsto_{Decompose_{\mathcal{K}}} & (X \ll \mathtt{a} \wedge M \ll \mathtt{Empty})\ X \\
\mapsto\!\!\!\!\rightarrow & \mathtt{a}
\end{array}
$$

When we apply $\mathfrak{car}$ to Empty we obtain, as in O'CAML, a (run-time) error that is represented in the ρ_x-calculus by a matching constraint without solutions:

$$
\begin{array}{ll}
 & (\mathtt{Cons}(X, M) \twoheadrightarrow X)\ \mathtt{Empty} \\
\mapsto_{\rho} & (\mathtt{Cons}(X, M) \ll \mathtt{Empty})\ X
\end{array}
$$

Unlike in O'CAML, the matching failure is explicit and the programmer can have a better understanding of the error. Actually, the O'CAML interpreter answers to the previous definitions of car and cdr with:

```
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: []
```

In O'CAML, the matching failures are treated at the meta-level whereas this is done automatically in the ρ-calculus thanks to the reduction semantics.

The next example illustrates the usefulness of explicit constraint application when we want to track the source (cause) of the failure.

Example 3.7 (Run-time error: matching failure)
Let us consider the following rule that checks if two persons are brothers, *i.e.* if they have the same father:

$\mathtt{Brother}(\mathtt{Person}(\mathtt{Name}(X),\mathtt{Father}(Z)),\mathtt{Person}(\mathtt{Name}(Y),\mathtt{Father}(Z))) \twoheadrightarrow \mathtt{tt}$

When checking if two concrete persons (Alice and Bob)are brothers by applying this rule to the corresponding term:

$\mathtt{Brother}(\mathtt{Person}(\mathtt{Name}(\mathtt{Alice}),\mathtt{Father}(\mathtt{John})),\ \mathtt{Person}(\mathtt{Name}(\mathtt{Bob}),\mathtt{Father}(\mathtt{Jim})))$ we obtain as result the term

$(Z \ll \mathtt{John} \wedge Z \ll \mathtt{Jim})\ \mathtt{tt}$

indicating that the variable Z corresponding to the father cannot be instanti-

ated correctly, *i.e.* that the father of the two persons is not the same.

In a classical (non-explicit) approach the result would be

$\Big($Brother(Person(Name(X),Father(Z)),Person(Name(Y),Father(Z))) $\ll$ Brother(Person(Name(Alice),Father(John)), Person(Name(Bob),Father(Jim)))$\Big)$ tt

that is obviously more difficult to understand.

4 Properties

We present here the properties of the calculus and of its sub-calculi. We concentrate on the confluence, the basic property for implementations: the strategy used to apply the evaluation rules of the $\rho_{\mathtt{x}}$-calculus (Figure 2) has no consequences on the result. We first present properties on the explicit part of the calculus and then we use these results for the confluence of the calculus. The complete proofs can be found in [13].

The confluence of higher-order systems dealing with non-linear matching is still a difficult task because we do not know how to prevent non-joinable critical pairs as those coined for the first time by Klop [16] and therefore, in this paper, we will only consider *linear patterns* so as not to lose confluence.

4.1 Properties of the explicit part

One important property of the explicit calculus is the conservativity. This property shows that a reduction in the ρ-calculus can be simulated in the $\rho_{\mathtt{x}}$-calculus. This property is sometimes called simulation:

Lemma 4.1 (Conservativity)
For all terms A and B containing only first-order patterns[2] *and such that $A \mapsto_{\rho} B$, we have $A \mapsto\!\!\!\!\rightarrow_{\rho_x} B$.*

From this lemma, we can deduce the preservation of the confluence of the ρ-calculus by the $\rho_{\mathtt{x}}$-calculus. Since we do not deal with composition and meta-variables we also conjecture that the $\rho_{\mathtt{x}}$-calculus preserves strong normalization.

To show the confluence of the $\rho_{\mathtt{x}}$-calculus, we show first the strong normalization of the explicit constraint handling part of the calculus.

Lemma 4.2 *The relation $\mapsto_{\kappa}$ is strongly normalizing.*

Proof First of all, we define a measure ζ on terms representing the number of different variables in the left-hand side of constraint matching equations - but

[2] The restriction to first-order patterns is due to the restriction to first-order matching. Recall discussion in paragraph "Constraint computation" of Section 2.

not of substitutions. For example, X is not considered to be in a left hand side of a constraint matching equation in $\{X \ll A\}B$ whereas this is the case in $(X \ll A)B$. We show that $\mapsto_\kappa$ is strongly normalizing using the lexicographic product of ζ and $\succcurlyeq$ where $\succcurlyeq$ is the recursive path ordering induced by the precedence $\succ$: $\ll \succ \wedge$ and $\{\} \succ$; and $\{\} \succ$ "the application operator" and $\{\} \succ \twoheadrightarrow$ and $\{\} \succ \ll$ and $\{\} \succ \wedge$ and with the status "multiset" for the symbol $\{\}$. □

Lemma 4.3 *The relation $\mapsto_\kappa$ is locally confluent.*

Proof $\mapsto_\sigma$ is convergent since $\mapsto_\sigma$ is terminating as a sub-relation of $\mapsto_\kappa$ and confluent as an orthogonal and left-linear system. We denote by $\downarrow_\sigma(A)$ the normal form of A *w.r.t.* the rewrite system σ and we show the following substitution lemma:

For all terms A, B, C such that Y belongs to the set of free variables of A:

$$\downarrow_\sigma(\{X \ll A\}(\{Y \ll B\}C)) \ = \downarrow_\sigma(\{Y \ll \{X \ll A\}B\}(\{X \ll A\}C))$$

The substitution lemma is a consequence of the relationship between explicit and pure substitutions: we can show that the behavior of the explicit substitution $\{X \ll A\}B$ behaves (*i.e.* reduces) exactly like the meta-substitution (that handles variable capture). The proof is then done by analyzing all critical pairs which can easily be shown joinable as a consequence of the substitution lemma. □

4.2 Confluence of the calculus for linear patterns

The proof of confluence is based on Yokouchi's lemma. We apply it by choosing the relations $\mapsto_\kappa$ and $\mapsto_{\rho\delta_\parallel}$, where $\mapsto_{\rho\delta_\parallel}$ is the parallelization of ρ and δ. So, we need to prove the convergence of $\mapsto_\kappa$ (already done), the strong confluence of $\mapsto_{\rho\delta_\parallel}$ and the coherence diagram between the two relations. The lemma cannot be applied simply by taking the relation $\mapsto_{\rho\delta}$ since this relation does not have the diamond property.

Lemma 4.4 (Yokouchi's diagram) *For all terms A, B, C, if $A \mapsto_\kappa B$ and $A \mapsto_{\rho\delta_\parallel} C$ then there exists a term D such that $B \mapsto\!\!\!\!\twoheadrightarrow_\kappa \mapsto_{\rho\delta_\parallel} \mapsto\!\!\!\!\twoheadrightarrow_\kappa D$ and $C \mapsto\!\!\!\!\twoheadrightarrow_\kappa D$.*

Proof When the two steps from A to B and from A to C do not overlap, the lemma is easy. So we have to inspect every critical pair. As in the $\lambda\sigma_{\Uparrow}$-calculus a critical pair has a sense slightly different from the standard one because of the parallel reduction. Since a strict subexpression of a $\rho\delta_\parallel$ redex can never overlap with a κ redex, it is sufficient to work by cases on the derivation from A to B. □

Theorem 4.5 (Confluence)
The ρ_x-calculus is confluent for linear patterns.

Proof We have proved all the hypotheses of Yokouchi's lemma:

- $\mapsto_\kappa$ is strongly normalizing (Lemma 4.2).
- $\mapsto_\kappa$ is locally confluent (Lemma 4.3).
- $\mapsto_{\rho\delta_{\|}}$ is strongly confluent (parallelization of a linear rewrite system without any critical pairs).
- $\mapsto_\kappa$ and $\mapsto_{\rho\delta_{\|}}$ verify Yokouchi's diagram (Lemma 4.4).

Thus, we obtain that $\mapsto\!\!\!\!\!\rightarrow_\kappa \mapsto_{\rho\delta_{\|}} \mapsto\!\!\!\!\!\rightarrow_\kappa$ is confluent. To conclude the proof it is sufficient to remark that $\mapsto_{\rho_x} \subseteq \mapsto\!\!\!\!\!\rightarrow_\kappa \mapsto_{\rho\delta_{\|}} \mapsto\!\!\!\!\!\rightarrow_\kappa \subseteq \mapsto\!\!\!\!\!\rightarrow_{\rho_x}$. □

5 Extensions

In practice, our calculus is restrictive and one wants to extend it for expressiveness reasons (more matching theories) and efficiency reasons (composition of substitutions). In this section, we will briefly discuss the two points.

We only handled the case of syntactic matching for the defined symbols and we can see this as a drawback of our calculus. In practice, it is interesting to have the possibility to reason modulo (equational) theories *w.r.t.* the defined constants. This can be done by adjusting the part of the calculus dealing with explicit matching.

The ($Decompose_{\mathcal{K}}$) rule can be adapted according to the theory one wants to deal with. For example, if we want to deal with commutative symbols (not necessarily binary) we obtain the rule ($Decompose^C_{\mathcal{K}}$) - where $\mathfrak{S}_n$ denotes the permutations of $\{1, \ldots, n\}$:

$$Decompose^C_{\mathcal{K}}\ f(A_1, \ldots, A_n) \ll_C f(A'_1, \ldots, A'_n) \rightarrow \underset{\varphi \in \mathfrak{S}_n}{;} \left(\bigwedge_{i=1}^{n} A_i \ll_C A'_{\varphi(i)} \right)$$

If one prefers the AC (associative-commutative) matching theory, the corresponding decomposition rule should be specified.

In most of the applications, the empty theory for the structure ";" is not sufficient. For example, if one wants to encode multisets, the AC theory is useful. If one wants to encode rewrite systems in the ρ-calculus, one needs a special theory for the structure so as to erase matching failures (this theory is presented in [9]). In such cases, the confluence of the calculus must be carefully examined.

As far as it concerns efficiency, substitutions are identified, in this paper, to constraints of the form $X \ll A$. One of the main drawbacks of this approach

is the lack of a mechanism to merge different term traversals into a single one. This mechanism is proved to have an important impact on performance (see for example [18]) and is used in the implementation of SML. For example, if we denote $g^n(f(X,Y)) \triangleq g(g(\ldots g(f(X,Y))))$ then

$$\begin{aligned}
&(X \ll A \wedge Y \ll B)\ g^n(f(X,Y)) \\
\mapsto_{\rho_x}\ &(X \ll A)\{Y \ll B\}g^n(f(X,Y)) \\
\mapsto\!\!\!\!\to_{\rho_x}\ &(X \ll A)g^n(f(X,B)) && \text{first traversal} \\
\mapsto_{\rho_x}\ &\{X \ll A\}g^n(f(X,B)) \\
\mapsto\!\!\!\!\to_{\rho_x}\ &g^n(f(A,B)) && \text{second traversal}
\end{aligned}$$

So we need to traverse twice the term (that can be as big as wanted) to apply this very simple substitution. In fact, we want to visit the term not twice but only once, that is, we want to handle composition.

Of course, since one wants to identify solvable constraints, one needs to propose a nice way to label parts of constraints which are solvable and independent of the remaining constraints. Instead of "pushing out" of a constraint a sub-part of the form $X \ll A$ we do the same for labeled constraints. For this, the following composition rule should be added:

$$(Compose)\ \{\mathcal{C}\}(\{\mathcal{D}\}A) \rightarrow \{\mathcal{C} \wedge \{\mathcal{C}\}\mathcal{D}\}\ A$$

where $\mathcal{C}$ and $\mathcal{D}$ are constraints known as solvable. The study of this extension is under way.

Conclusion and future work

We have proposed a ρ-calculus of explicit constraint application well-suited to deal with errors. We have proved that it enjoys the classical properties of such a formalism, *i.e.*, the confluence of the calculus and the termination of the constraint handling part. We have seen that the calculus is really modular and can be adapted to many matching theories for the defined constants and for the structure operator ";". We can either choose to be atomic and give a simple definition of substitutions, or more general and efficient and define a calculus that handles substitution composition.

ρ-calculi and especially the $\rho_{\mathbf{x}}$-calculus, are new frameworks that can be seen as theoretical foundations for a new family of programming languages. Different extensions/variations of the ρ-calculus are now available: in [19] an imperative version of the calculus has been proposed and in [14] exceptions in the ρ-calculus were studied. One can mention that the ρ-calculus allows one

to design extremely powerful type systems such as those presented in [8].

Related work: In [3], a calculus called the PSA-Calculus was introduced. The explicit application of a rewrite rule and the *explicit matching handling* were coined for the first time in this ancestor of the ρ-calculus. Nevertheless, it was a first approach to make explicit rewriting and thus this calculus is really less powerful than the current ρ-calculus. For example, the PSA-Calculus is not powerful enough to allow strategies as explicit objects and thus there is a hierarchy between rules and strategies.

A rewriting calculus with explicit substitutions has been already proposed in [6]. This calculus is mainly an extension of the $\lambda\sigma$-calculus and is called the $\rho\sigma$-calculus. The approach is less general than the one presented here since this calculus makes explicit the substitution application but not the computations to go from constraints to substitutions. In [22], Nguyen studied a cooperation Coq-ELAN to automate proof assistants where the $\rho\sigma$-calculus have been intensively used to represent proof terms of rewrite derivations. The explicit treatment of matching in the $\rho_{\mathtt{x}}$-calculus should be a useful tool to obtain normalization traces in some non-trivial matching theories.

Future work: Different extensions should be studied before an implementation can be realized:

- To handle substitution composition. Actually, the ability to merge different structure traversals into one has an important impact on performance as shown in [18].
- To deal with α-conversion. One possible approach is to follow the work about the λ-calculus [15] or to use deBruijn [10] indices.
- To propose a powerful named exception mechanism, by taking advantage of the very general management of errors.

More generally, we want to understand what an interpreter/compiler for the ρ-calculus could mean and how to implement it. This question is strongly related to our intend to study integrated programming and proving environments where computations and deductions are uniformly integrated, *i.e.*, to unify functional and rewriting based languages (*e.g.*, ML, ELAN, Maude), proof assistants and theorem provers (*e.g.*, Coq, Isabelle, PVS, ...).

References

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

[2] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. North Holland, 1984. Second edition.

[3] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In *International Journal of Foundations of Computer Science* , 2001.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In *Proc. of WRLA'98*, volume 15 of *ENTCS*, September 1998.

[5] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proc. of WRLA'96*, volume 4 of *ENTCS*, 1996.

[6] H. Cirstea. *Calcul de réécriture : fondements et applications.* Thèse de Doctorat, Université Henri Poincaré – Nancy 1, France, October 2000.

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

[8] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting calculus with(out) types. In *Proc. of WRLA'02*, volume 71 of *ENTCS*, September 2002.

[9] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In *Proc. of WRS'03*, volume 83 of *ENTCS*, June 2003.

[10] N. G. de Bruijn. Lambda-calculus with name free formulas involving symbols that represent reference transforming mappings. *Indagationes Mathematicae*, 40:348–356, 1978.

[11] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In *Information and Computation*, 157(1/2):183–235, 2000.

[12] S. Eker. Associative-Commutative Matching Via Bipartite Graph Matching. *The Computer Journal*, 38(5):381–399, 1995.

[13] G. Faure. Explicit rewriting calculus. Master thesis, IRISA, LORIA, ENS-Cachan, September 2003.

[14] G. Faure and C. Kirchner. Exceptions in the rewriting calculus. In *Proc. of RTA'02*, volume 2378 of *LNCS*, pages 66–82, July 2002.

[15] D. Hendriks and V. van Oostrom. The adbmal-calculus. In *Proc. of CADE'03*, volume 2741 of *LNAI*, pages 136–150, 2003.

[16] J. Klop. *Combinatory Reduction Systems.* Ph.D. thesis, Mathematisch Centrum - Amsterdam, Holland, 1980.

[17] P. Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In *Conference Record of POPL '94*, pages 60–69. ACM, January 1994.

[18] C. Liang, G. Nadathur, and X. Qi. Choices in representation and reduction strategies for lambda terms in intensional contexts. Technical Report 2003/2, University of Minnesota, October 2003.

[19] L. Liquori and B.P. Serpette. An imperative rewriting calculus. In *Proc. of PPDP'04*, *LNCS*, August 2004.

[20] P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In *Proc. of CC'03*, volume 2622 of *LNCS*, pages 61–76, May 2003.

[21] C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types.* Thèse de doctorat, Université Paris 7, Novembre 1997.

[22] Q.-H. Nguyen. Certifying Term Rewriting Proof in `ELAN`. In *Proc. of RULE'01*, volume 59 of *ENTCS*, September 2001.

[23] K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, University of Copenhagen, Denmark, February 1996.

[24] A. Stump, A. Deivanayagam, S. Kathol, D. Lingelbach, and D. Schobel. Rogue decision procedures. In *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2003.

[25] E. Chailloux, P. Manoury, and B. Pagano *Développement d'applications avec Objective Caml.* Editions O'Reilly. 2000.