Matching Power

Horatiu Cirstea, Claude Kirchner, and Luigi Liquori

LORIA INRIA INPL ENSMN
54506 Vandoeuvre-lès-Nancy BP 239 Cedex France
{Horatiu.Cirstea,Claude.Kirchner,Luigi.Liquori}@loria.fr
www.loria.fr/{~cirstea,~ckirchne,~lliquori}

Abstract. In this paper we give a simple and uniform presentation of the rewriting calculus, also called *Rho Calculus*. In addition to its simplicity, this formulation explicitly allows us to encode complex structures such as lists, sets, and objects. We provide extensive examples of the calculus, and we focus on its ability to represent some object oriented calculi, namely the *Lambda Calculus of Objects* of Fisher, Honsell, and Mitchell, and the *Object Calculus* of Abadi and Cardelli. Furthermore, the calculus allows us to get object oriented constructions unreachable in other calculi. *In summa*, we intend to show that because of its matching ability, the Rho Calculus represents a *lingua franca* to naturally encode many paradigms of computations. This enlightens the capabilities of the rewriting calculus based language ELAN to be used as a logical as well as powerful semantical framework.

1 Introduction

Matching is a feature provided implicitly in many, and explicitly in few, programming languages. In this paper, by making matching a "first-class" concept, we present, experiment with, and show the expressive power of a new version of the rewriting calculus, also called Rho Calculus (ρCal).

The ability to discriminate patterns is one of the main basic mechanisms the human reasoning is based on; as one commonly says "one picture is better than a thousand explanations". Indeed, the ability to recognize patterns, *i.e.* pattern matching, is present since the beginning of information processing modeling. Instances of it can be traced back to pattern recognition and it has been extensively studied when dealing with strings [26], trees [19] or feature objects [2].

Matching occurs implicitly in many languages through the parameter passing mechanism but often as a very simple instance, and explicitly in languages like PROLOG and ML, where it can be quite sophisticated [28,27]. It is somewhat astonishing that one of the most commonly used model of computation, the lambda calculus, uses only trivial pattern matching. This has been extended, initially for programming concerns, either by the introduction of patterns in lambda calculi [33,36,11], or by the introduction of matching and rewrite rules in functional programming languages. And indeed, many works address the integration of term rewriting with lambda calculus, either by enriching first-order rewriting with higher-order capabilities, or by adding to lambda 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 [25] and other higher-order rewriting systems [39,29], in the second case the works on combination of lambda calculus with term rewriting [30,4,16,22], to mention only a few.

Embedding more information in the matching process makes it appropriate to deal with complex tasks like program transformations [20] or theorem proving [32]. In that direction, matching in elaborated theories has been also studied extensively, either in equational theories [21,5] or in higher-order logic [12,31], where it is still an open problem at order five.

Matching allows one to discriminate between alternatives. Once the patterns are recognized, the action to be taken on the appropriate pattern should be described, and this is what rewriting is designed for. The corresponding pattern is thus rewritten in an appropriate instance of a new one. The mechanism that describes this process is the *rewriting calculus*. Its main design concept is to make all the basic ingredients of rewriting explicit objects, in particular the notions of *rule application* and *result*. By making the application explicit, the calculus emphasizes on one hand the fundamental role of matching, and on the other hand the intrinsic higher-order nature of rewriting. By making the results explicit, the Rho Calculus has the ability to handle non-determinism in the sense of a collection of results: an empty collection of results represents an application failure, a singleton represents a deterministic result, and a collection with more than one element represents a non-deterministic choice between the elements of the collection.

For example, assuming a, b, and c to be different constants, in the rewriting calculus the application of the rewrite rule $a \to b$ on the term a is expressed by the term $(a \to b) \bullet a$. This term is evaluated into the term b. The application of same rule on the term c is expressed by the term $(a \to b) \bullet c$ and it evaluated to *null*, therefore memorizing the fact that the rule is not applicable since the term a does not match the term c. Of course one can use variables and the simplest way to do it is just with rewrite rules whose left-hand side is a single variable term, like in $(X \to b) \bullet c$. Such a rule always applies and the previous term evaluates to c. This trivial rule application corresponds indeed exactly to the lambda calculus application: the previous term could be written as $(\lambda X.b)\, c$. This enlightens one of the nice feature of the rewriting calculus to abstract not only on variables like in the previous example, but also on arbitrary terms, including non-linear ones.

This matching power of the calculus provides important expressivity capabilities. As suggested by the previous example, it embeds lambda calculus, but also permits the representation of term rewrite derivations, even in the conditional case. More generally, it allows us to describe traversal, evaluation and search strategies like leftmost innermost, or breath first [7]. In [7], we have shown that *conditional* rewrite rules of the form "$l \to r$ if c" can be faithfully represented in the rewriting calculus as $l \to (\mathsf{True} \to r) \bullet (strat \bullet c)$, where *strat* is a suitable term representing the normalization strategy of the condition.

Rewriting is central in several programming languages developed since the seventies. Amongst the main ones let us mention OBJ [18], ASF+SDF [35], Maude [10], CafeOBJ [15], Stratego [38], and ELAN [24,3,34] which has been at the origin of some of the main concepts of the rewriting calculus. In turn, the Rho Calculus provides a natural semantics to such languages, and in particular to ELAN, covering the notion of rule application strategy, a fundamental concept of the language.

In this paper, we give the newest description of the Rho Calculus, as introduced in [9]. It provides a simplified version of the evaluation rules of the calculus as well as a generic and explicit handling of result structures, a point left open in the previous works [6,7].

The contributions of this paper are therefore the following:

- we provide a broad set of examples showing the expressiveness of the Rho Calculus obtained mainly thanks to its "matching power" and how this makes it suitable to uniformly model various paradigms of computation;
- we show how the matching power of the Rho Calculus allows us to encode two major object-calculi which have strongly influenced the type-theoretical research of the last five years: the *Object Calculus* ($\varsigma\mathcal{O}bj$) of Abadi and Cardelli [1] and the *Lambda Calculus of Objects* of Fisher, Honsell, and Mitchell [14] ($\lambda\mathcal{O}bj$). Moreover, we show two examples in Rho Calculus that cannot be encoded in the above calculi.

Road Map of the Paper. The paper is structured as follows: in Section 2, we present the syntax and the small-step semantics of the Rho Calculus; Section 3 presents a *plethora* of examples describing the power of matching; Section 4 presents the encoding of the Lambda Calculus of Objects and of the Object Calculus in the Rho Calculus. Conclusions and further works are finally discussed in Section 5. An extended version of the paper can be found in [8].

2 Syntax and Semantics

Notational Conventions. In this paper, the symbol t ranges over the set $\mathcal{T}$ of terms, the symbols $S, X, Y, Z, \dots$ range over the infinite set $\mathcal{V}$ of variables, the symbols $null, \oplus, \circ, a, b, \dots, z, 0, 1, 2, \dots$ range over the infinite set $\mathcal{C}$ of constants of fixed arity. All symbols can be indexed. The symbol $\equiv$ denotes syntactic identity of objects like terms or substitutions. We work modulo α-conversion, and we follow the Barendregt convention that free and bound variables have different names.

2.1 Syntax

The syntax of the ρCal is defined as follows:

$$\begin{array}{rll} t ::= & a \mid X \mid t \to t \mid t \bullet t \mid & \text{plain terms} \\ & null \mid t, t & \text{structured terms} \end{array}$$

The main intuition behind this syntax is that a rewrite rule $t \to t$ is an *abstraction*, the left-hand-side of which determines the bound variables and some pattern structure. The application of a ρCal-term on another ρCal-term is represented by "$\bullet$". The terms can be grouped together into a structure built using the "," operator and, according to the theory behind this operator, different structures can be obtained. The term *null* denotes an empty structure.

We assume that the application operator "$\bullet$" associates to the left, while the "$\to$" and the "," operators associate to the right. The priority of the application "$\bullet$" is higher than that of the "$\to$" operator which is in turn of higher priority than the "," operator.

Definition 1 (Some Type Signatures and Abbreviations).

$$\begin{array}{llllll}
\to : \mathcal{T} \times \mathcal{T} \Rightarrow \mathcal{T} & & t_1.t_2 & \triangleq t_1\bullet t_2\bullet t_1 & \textit{self-application} \\
\bullet\ : \mathcal{T} \times \mathcal{T} \Rightarrow \mathcal{T} & \textit{and} & t(t_1 \ldots t_n) & \triangleq t\bullet t_1 \ldots \bullet t_n & \textit{function-application}\ (n \in \mathbb{N}) \\
,\ : \mathcal{T} \times \mathcal{T} \Rightarrow \mathcal{T} & & (t_i)^{i=1\ldots n} & \triangleq t_1, \ldots, t_n & \textit{structure}\ (n \in \mathbb{N})
\end{array}$$

We draw the attention of the reader on the main difference between "$\bullet$" denoting the *application*, and "." denoting the object-oriented *self-application* operator.

2.2 Matching Theories

An important parameter of the ρCal is the matching theory $\mathbb{T}$. We give below examples of theories $\mathbb{T}$ defined equationally.

Definition 2 (Matching theories).

- *the Empty theory $\mathbb{T}_\emptyset$ of equality (up to α-conversion) is defined as the following inference rules:*

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ (Tra) \qquad \frac{t_1 = t_2}{t_2 = t_1}\ (Sym) \qquad \frac{t_1 = t_2}{t_3[t_1]_p = t_3[t_2]_p}\ (Ctx) \qquad \frac{}{t = t}\ (Ref)$$

  where $t_1[t_2]_p$ denotes the term t_1 with the term t_2 at position p. The α-conversion definition follows the standard intuition and is made precise for ρCal in [7].
- *the theory of Commutativity $\mathbb{T}_{C(f)}$ (resp. Associativity $\mathbb{T}_{A(f)}$) is defined as $\mathbb{T}_\emptyset$ plus the following inference rules:*

$$\frac{}{f(t_1\ t_2) = f(t_2\ t_1)}\ (Com) \qquad \frac{}{f(f(t_1\ t_2)\ t_3) = f(t_1\ f(t_2\ t_3))}\ (Ass)$$

- *the theory of Idempotency $\mathbb{T}_{I(f)}$ is defined as $\mathbb{T}_\emptyset$ plus the axiom $f(t\ t) = t$.*
- *the theory of Neutral Element $\mathbb{T}_{N(f^0)}$ is defined as $\mathbb{T}_\emptyset$ plus the following inference rules:*

$$\frac{}{f(0\ t) = t}\ (0_{Left}) \qquad \frac{}{f(t\ 0) = t}\ (0_{Right})$$

- *the theory of the Lambda Calculus of Objects, $\mathbb{T}_{\lambda Obj}$, is obtained by considering the symbol "," as associative and null as its neutral element, i.e.:*

$$\mathbb{T}_{\lambda Obj} = \mathbb{T}_{A(,)} \cup \mathbb{T}_{N(,^{null})}$$

– *the theory of the Object Calculus, $\mathbb{T}_{\varsigma Obj}$, is obtained by considering the symbol "," as associative and commutative and null as its neutral element, i.e.:*

$$\mathbb{T}_{\varsigma Obj} = \mathbb{T}_{A(,)} \cup \mathbb{T}_{C(,)} \cup \mathbb{T}_{N(,null)} = \mathbb{T}_{\lambda Obj} \cup \mathbb{T}_{C(,)}$$

Other interesting theories can be built from the above ones, such as *e.g.* $\mathbb{T}_{MSet(f,nil)}$, and $\mathbb{T}_{Set(f,nil)}$ [7]. For the sake of completeness, we include in the paper the definition of syntactic matching, which can also be found in [7], together with more explanatory examples.

Definition 3 (Syntactic Matching). *For a given theory $\mathbb{T}$ over ρCal-terms:*

1. *a* $\mathbb{T}$-match equation *is a formula of the form $t_1 \ll_{\mathbb{T}} t_2$;*
2. *a* substitution σ *is a solution of the $\mathbb{T}$-match equation $t_1 \ll_{\mathbb{T}} t_2$ if $\sigma t_1 =_{\mathbb{T}} t_2$;*
3. *a* $\mathbb{T}$-matching system *is a conjunction of $\mathbb{T}$-match equations;*
4. *a substitution σ is a solution of a $\mathbb{T}$-matching system if it is a solution of all the $\mathbb{T}$-match equations in it;*
5. *a $\mathbb{T}$-matching system is* trivial *when all substitutions are solution of it and we denote by $\mathbb{F}$ a $\mathbb{T}$-matching system without solution;*
6. *we define the function $\mathcal{S}ol$ on a $\mathbb{T}$-matching system T as returning the $\prec$-ordered[1] list of all $\mathbb{T}$-matches of T when T is not trivial and the list containing only $\sigma_{\mathbf{id}}$, where $\sigma_{\mathbf{id}}$ is the identity substitution, when T is trivial.*

Notice that when the matching algorithm fails (*i.e.* returns $\mathbb{F}$), the function $\mathcal{S}ol$ returns the empty list. A more detailed discussion on decidability of matching can be found in [7].

For example, in $\mathbb{T}_{\emptyset}$, the matching substitution from a ρCal-term t_1 to a ρCal-term t_2 can be computed by the rewrite system presented in Figure 1, where the symbol $\wedge$ is assumed to be associative and commutative, and $\diamond_1$, $\diamond_2$ are either constant symbols or the prefix notations of "," or "•" or "$\rightarrow$".

Starting from a matching system T, the application of this rule set terminates and returns either $\mathbb{F}$ when there are no substitutions solving the system, or a system T' in "normal form" from which the solution can be trivially inferred [23]. This set of rules could be extended to deal with more elaborated theories like commutativity.

2.3 Operational Semantics

For a given total ordering $\prec$ on substitutions (which is left implicit in the notation) and a theory $\mathbb{T}$, the operational semantics is defined by the computational rules given in Figure 2. The central idea of the main rule of the calculus (ρ) is that the application of a rewrite rule $t_1 \rightarrow t_2$ at the root position of a term t_3, consists in computing all the solutions of the matching equation ($t_1 \ll_{\mathbb{T}} t_3$) in the theory $\mathbb{T}$ and applying all the substitutions from the $\prec$-ordered list returned by the function $\mathcal{S}ol(t_1 \ll_{\mathbb{T}} t_3)$ to the term t_2. When there

[1] We consider a total order $\prec$ on the set of substitutions [7].

$$\begin{array}{lll}
\diamond_1(t_1 \ldots t_n) \ll_{\mathbb{T}_\emptyset} \diamond_2(t'_1 \ldots t'_m) & \leadsto & \begin{cases} \bigwedge\limits_{i=1\ldots n} t_i \ll_{\mathbb{T}_\emptyset} t'_i & \text{if } \diamond_1 \equiv \diamond_2 \text{ and } n = m \\ \mathbb{F} & \text{otherwise} \end{cases} \\
(X \ll_{\mathbb{T}_\emptyset} t) \wedge (X \ll_{\mathbb{T}_\emptyset} t') & \leadsto & \begin{cases} X \ll_{\mathbb{T}_\emptyset} t & \text{if } t =_{\mathbb{T}_\emptyset} t' \\ \mathbb{F} & \text{otherwise} \end{cases} \\
t \ll_{\mathbb{T}_\emptyset} X & \leadsto & \mathbb{F} \qquad\qquad\qquad\quad \text{if } t \notin \mathcal{V} \\
\mathbb{F} \wedge (t \ll_{\mathbb{T}_\emptyset} t') & \leadsto & \mathbb{F}
\end{array}$$

Fig. 1. Rules for Syntactic Matching

$$\begin{array}{llll}
(\rho) & (t_1 \to t_2)\bullet t_3 & \mapsto_{\mathbb{T}} & \begin{cases} null & \text{if } t_1 \ll_{\mathbb{T}} t_3 \text{ has no solution} \\ \sigma_1 t_2, \ldots, \sigma_n t_2 & \text{if } \sigma_i \in \mathcal{S}ol(t_1 \ll_{\mathbb{T}} t_3), \sigma_1 \prec \sigma_{i+1},\ n \leq \infty \end{cases} \\
(\epsilon) & (t_1, t_2)\bullet t_3 & \mapsto_{\mathbb{T}} & t_1\bullet t_3, t_2\bullet t_3 \\
(\nu) & null\bullet t & \mapsto_{\mathbb{T}} & null
\end{array}$$

Fig. 2. Evaluation rules of the ρCal

is no solution for the matching equation $t_1 \ll_{\mathbb{T}} t_3$, the special constant *null* is obtained as result of the application. Notice that in some theories, there could be an infinite set of solutions to the matching problem $(t_1 \ll_{\mathbb{T}} t_3)$; possible ways to deal with the infinitary case are described in [6].

The other rules (ϵ) and (ν) deal with the distributivity of the application on the structures whose constructors are "," and *null*. When the theory $\mathbb{T}$ is clear from the context, its denotation will be omitted. Notice that if t_1 is a variable, then the (ρ)-rule corresponds exactly to the (β)-rule of the lambda calculus.

With respect to the previous presentation of the Rho Calculus [7], we have modified the notation of the application operator which was denoted $[\_](\_)$, but more importantly, the evaluation rules have been simplified on one hand, and generalized to deal with generic result structures on the other hand.

As usual, given a theory $\mathbb{T}$, we denote by $=_\rho$ the smallest reflexive, symmetric, and transitive relation containing $\mapsto_{\mathbb{T}}$, stable by context and substitution. When working modulo reasonably powerful theories $\mathbb{T}$, the evaluation rules of the ρCal are confluent:

Theorem 1 (Confluence in $\mathbb{T}_\emptyset$). *Given a term t_1 such that all its abstractions contain no arrow in the first argument, if $t_1 \mapsto\!\!\!\!\rightarrow_{\mathbb{T}_\emptyset} t_2$ and $t_1 \mapsto\!\!\!\!\rightarrow_{\mathbb{T}_\emptyset} t_3$ then there exists a term t_4 such that $t_2 \mapsto\!\!\!\!\rightarrow_{\mathbb{T}_\emptyset} t_4$ and $t_3 \mapsto\!\!\!\!\rightarrow_{\mathbb{T}_\emptyset} t_4$.*

Proof. The proof follows the same lines defined in [7].

3 Examples in Rho Calculus

In the following section we present some simple examples intended to help the reader in the understanding of the behavior of the Rho Calculus.

Example 1 (In $\mathbb{T}_\emptyset$).

1. The application of the simple rewrite rule $a \to b$ to a, *i.e.* $(a \to b)\bullet a$, is evaluated to b since $Sol(a \ll_{\mathbb{T}_\emptyset} a) = \sigma_{\mathbf{id}}$ and $\sigma_{\mathbf{id}} b \equiv b$;
2. The matching between the left-hand side of the rule and the argument can also fail and in this case the result of the application is the constant *null*, *i.e.*: $(a \to b)\bullet c \overset{\rho}{\mapsto} null$;
3. When the left-hand side of a rewrite rule is not a ground term, the matching can yield a substitution different from $\sigma_{\mathbf{id}}$, e.g. $(X \to X)\bullet a \overset{\rho}{\mapsto} [X/a]X \equiv a$;
4. The non-deterministic application of two rewrite rules is represented by the application of the structure containing the respective rules: $(X \to X(a), Y \to Y(b))\bullet c \overset{\epsilon}{\mapsto} (X \to X(a))\bullet c, (Y \to Y(b))\bullet c \overset{\rho}{\mapsto\!\!\!\!\to} [X/c]X(a), [Y/c]Y(b) \equiv c(a), c(b)$;
5. The selection of the field cx inside the record structure $(cx \to 0, cy \to 0)$ evaluates to the term $(0, null)$, *i.e.*: $(cx \to 0, cy \to 0)\bullet cx \overset{\epsilon}{\mapsto} (cx \to 0)\bullet cx, (cy \to 0)\bullet cx \overset{\rho}{\mapsto\!\!\!\!\to} (0, null)$;
6. Functions are first-class entities in the ρCal: $(X \to (X\bullet a))\bullet\ (Y \to Y) \overset{\rho}{\mapsto} (Y \to Y)\bullet a \overset{\rho}{\mapsto} a$;
7. The lambda calculus with *patterns* [33] can be easily represented in the ρCal. For instance, the lambda-term $\lambda Pair(X\,Y).X$, can be represented and reduced as follows: $(Pair(X\ Y) \to X)\bullet Pair(a\ b) \overset{\rho}{\mapsto} [X/a, Y/b]X \equiv a$;
8. Starting from the fixed-point combinators of the lambda calculus, we can define a ρCal-term that applies recursively a given ρCal-term. We use the classical fixed-point $Y_\lambda \triangleq (A_\lambda\ A_\lambda)$ with $A_\lambda \triangleq \lambda X.\lambda Y.Y(XXY)$, which can be translated as $Y_\rho \triangleq A_\rho\bullet A_\rho$ with $A_\rho \triangleq X \to Y \to Y\bullet(X\bullet X\bullet Y)$. Then: $Y_\rho\bullet t \triangleq A_\rho\bullet A_\rho\bullet t \triangleq (X \to Y \to Y\bullet(X\bullet X\bullet Y))\bullet A_\rho\bullet t \overset{\rho}{\mapsto} (Y \to Y\bullet(A_\rho\bullet A_\rho\bullet Y))\bullet t \overset{\rho}{\mapsto} t\bullet(A_\rho\bullet A_\rho\bullet t) \triangleq t\bullet(Y_\rho\bullet t)$. Starting from the Y_ρ term, we can define more elaborated terms describing, for example, the repeated application of a given term or normalization strategies according to a given rewrite rule [7];
9. Let $car \triangleq X, Y \to X$, $cdr \triangleq X, Y \to Y$, $cons \triangleq X \to Y \to (X, Y)$. It is easy to check that $car(a, b, c, null) \mapsto\!\!\!\!\to a$, and that $cdr(a, b, c, null) \mapsto\!\!\!\!\to b, c, null$, and that $cons(d\ a, b, c, null) \mapsto\!\!\!\!\to d, a, b, c, null$.

Example 2 (In $\mathbb{T}_A$, $\mathbb{T}_C$, $\mathbb{T}_{AC}$, and $\mathbb{T}_{N(f^0)}$).

1. ($\mathbb{T}_{A(\circ)}$) The application of the rewrite rule $\circ(X\ Y) \to X$ to $\circ(a\ \circ(b\ \circ(c\ d)))$ reduces, thanks to the associativity of $\circ$, to $(a, \circ(a\ b), \circ(a\ \circ(b\ c)))$;
2. ($\mathbb{T}_{C(\oplus)}$) The application of the rewrite rule $\oplus(X\ Y) \to X$ to $\oplus(a\ b)$ reduces, thanks to the associativity-commutativity of $\oplus$, to (a, b), a structure representing all possible results;
3. ($\mathbb{T}_{AC(\oplus)}$) The application of the rewrite rule $\oplus(X\ \oplus(X\ Y)) \to \oplus(X\ Y)$ to $\oplus(a\ \oplus(b\ \oplus(c\ \oplus(a\ d))))$ reduces to $\oplus(a\ \oplus(b\ \oplus(c\ d)))$. The search for the two equal elements is done by matching thanks to the associativity-commutativity of the $\oplus$ operator, while the elimination of doubles is performed by the rewrite rule;

4. ($\mathbb{T}_{N(f^0)}$) Using a theory with a neutral element allows us to "ignore" variables from the rewrite rules. For example, the rewrite rule $X \oplus a \oplus Y \to X \oplus b \oplus Y$ replaces an a with a b in a structure built using the "$\oplus$" operator and containing one or more elements. The application of the previous rewrite rule to $b \oplus a \oplus b$ reduces to $b \oplus b \oplus b$ and the same rule applied to a leads to b, since $a =_{\mathbb{T}_{N(\oplus^0)}} 0 \oplus a \oplus 0$.

The next example shows how the object oriented paradigm can be easily captured in the $\mathbb{T}_{\lambda\mathcal{O}bj}$. In particular we focus our example on the usage of the pseudo-variable `this` which is crucial for sending messages inside method bodies. In the *ρCal*, a method is seen as a term of the shape $m \to S \to t_m$, where m is the name of the method, S is a variable playing the role of `this` and t_m is the body of the method that can contain free occurrences of S. Sending a message m to a structure (*i.e.* an object) t is represented via the alias $t.m$, *i.e.* $t \bullet m \bullet t$. Intuitively, if the method m exists in the structure t, then its body t_m can be executed with the binding of S to the object itself. This type of application is also called, in the object-oriented jargon, *self-application*, and it is fundamental for modeling mutual recursion between methods inside an object.

Example 3 (In $\mathbb{T}_{\lambda\mathcal{O}bj}$).

1. This example presents a simple object t with only one method a that do not effectively use the variable S. Let $t \triangleq a \to S \to b$. Then, $t.a \triangleq t \bullet a \bullet t \overset{\rho}{\mapsto} (\sigma_{\mathbf{id}}(S \to b)) \bullet t \equiv (S \to b) \bullet t \overset{\rho}{\mapsto} b$;
2. This example presents an object t with a non-terminating method ω. Let $t \triangleq \omega \to S \to S.\omega$. Then, $t.\omega \overset{\rho}{\mapsto} (S \to S.\omega) \bullet t \overset{\rho}{\mapsto} t.\omega \mapsto\!\!\!\!\to \dots$;
3. We consider another object with a non-terminating behavior consisting of two methods *ping* and *pong*, one calling the other via the variable S. Let $t \triangleq (ping \to S \to S.pong, pong \to S \to S.ping)$. Then, $t.ping \triangleq t \bullet ping \bullet t \overset{\epsilon}{\mapsto} ((ping \to S \to S.pong) \bullet ping, (pong \to S \to S.ping) \bullet ping) \bullet t \overset{\rho}{\mapsto\!\!\!\!\to} (S \to S.pong) \bullet t, null =_{\mathbb{T}_{\lambda\mathcal{O}bj}} (S \to S.pong) \bullet t \overset{\rho}{\mapsto} t.pong \mapsto\!\!\!\!\to t.ping \mapsto\!\!\!\!\to \dots$

In the above example, we can notice how natural the use of matching is for directly selecting the method name. Starting from these simple examples, we can now imagine how matching can be use in its full generality (*i.e.* allowing variables as well as appropriate equational theories) in order to deal with more general objects and methods. The purpose of the rest of this paper is to make these aspects precise.

4 Object-Based in Rho Calculus

In this section we focus on two major object-calculi which have influenced the type-theoretical research of the last five years:

– The *Lambda Calculus of Objects* of Fisher, Honsell, and Mitchell [14];
– The *Object Calculus* of Abadi and Cardelli [1].

Both calculi are *prototype-based i.e.* they are based on the notion of "objects" and not of "classes". Nevertheless, classes can be easily encoded as suitable objects. Those calculi have been extensively studied in a "typed" setting where the main objective was to conceive sound type systems capturing the unfortunate run-time error `message-not-understood` which happen when we send a message m to an object which do not have the method m in its interface.

As previously shown in Example 3, structured-terms are well suited to represent objects and to model the special pseudovariable `this`. In order to support the intuition, we start by showing the way some classical examples of objects can be easily expressed in the ρCal.

Example 4 (A Point Object Encoding in $\mathbb{T}_{\lambda\mathcal{O}bj}$). Given the symbols val, get, set and v (used to denote pairs), an object $Point$ is encoded in ρCal by

$$val \to S \to v(1\ 1), get \to S \to S.val, set \to S \to v(X\ Y) \to (S, val \to S' \to v(X\ Y))$$

The term $Point$ represents an object with an attribute val and two methods get and set. The method get gives access to the attribute, while method set is used for modifying the attribute by adding the new value at the end of the object. In this context, it is easy to check that $Point.get \mapsto\!\!\!\!\twoheadrightarrow v(1\ 1)$, and $Point.set(v(2\ 2)) \mapsto\!\!\!\!\twoheadrightarrow Point, (val \to S' \to v(2\ 2))$, and $Point.set(v(2\ 2)).get \mapsto\!\!\!\!\twoheadrightarrow v(1\ 1), v(2\ 2)$. Worthy of notice is that:

1. The call $Point.set(v(2\ 2))$ produces a result which consists of the old $Point$ and the new (modified) value for the attribute val, *i.e.* $val \to S' \to v(2\ 2)$;
2. The call $Point.set(v(2\ 2)).get$ produces a structure composed of two elements, the former representing the value of val before the execution of set (*i.e.* before a side effect), and the latter one after the execution of set;
3. A trivial strategy to recover determinism is to consider only the last value from the list of results, *i.e.* $v(2\ 2)$. From this point of view, the ρCal can be also understood as a formalism to study side effects in imperative calculi;
4. A way to fix imperative features is to modify the encoding of the method set by considering the term $kill_n \triangleq (X, n \to Z, Y) \to X, Y$, and by defining the new object $Point_{imp}$ as

   $$val \to \ldots, get \to \ldots, set \to S \to v(X\ Y) \to (kill_{val}(S), val \to S' \to v(X\ Y))$$

   such that $Point_{imp}.get \mapsto\!\!\!\!\twoheadrightarrow v(1\ 1)$, and $Point_{imp}.set(v(2\ 2)) \mapsto\!\!\!\!\twoheadrightarrow val \to S' \to v(2\ 2), get \to \ldots, set \to \ldots$, and $Point_{imp}.set(v(2\ 2)).get \mapsto\!\!\!\!\twoheadrightarrow v(2\ 2)$;
5. The moral of this example is that the encoding of objects into the ρCal can strongly modify the behavior of a computation.

In the next example we present the encoding of the Fisher, Honsell, and Mitchell fixed-point operator [14] and its generalization in the ρCal.

Example 5 (A Fixed Point Object). Assume symbols rec and f. The fixed-point object Fix_f for f can be represented in the ρCal as $Fix_f \triangleq rec \to S \to f(S.rec)$. It is not hard to verify that $Fix_f.rec \mapsto\!\!\!\!\twoheadrightarrow f(Fix_f.rec)$. This fixed point can be generalized as $Fix \triangleq rec \to S \to X \to X(S.rec(X))$ and its behavior will be $Fix.rec(f) \mapsto\!\!\!\!\twoheadrightarrow f(Fix.rec(f))$.

4.1 The Lambda Calculus of Objects

We now present a translation of the Lambda Calculus of Objects of Fisher, Honsell, and Mitchell [14] into the ρCal. This calculus is an untyped lambda calculus with constants enriched with object primitives. A new object can be created by modifying and/or extending an existing prototype object; the result is a new object which inherits all the methods and fields of the prototype. This calculus is trivially computationally complete, since the lambda calculus is built in the calculus itself. The syntax and the small-step semantics of $\lambda\mathcal{O}bj$ we present in this paper are inspired by the work of [17].

Syntax and Operational Semantics. The syntax of the calculus is defined as follows:

$$\begin{aligned} M, N ::= {} & \lambda X.M \mid MN \mid X \mid c \mid \\ & \langle\,\rangle \mid \langle M \leftarrow n = N\rangle \mid \langle M \;\leftarrow\!\!+\, n = N\rangle \mid M \Leftarrow n \mid Sel(M, m, N) \end{aligned}$$

Let $\leftarrow\!*$ be either $\leftarrow$ or $\leftarrow\!\!+$; the small-step semantics is defined by

$$\begin{array}{llll} (\lambda X.M)\, N \mapsto_{\lambda\mathcal{O}bj} [X/N]M & & Sel(\langle M \leftarrow\!* n = N\rangle, n, P) & \mapsto_{\lambda\mathcal{O}bj} NP \\ M \Leftarrow m & \mapsto_{\lambda\mathcal{O}bj} Sel(M, m, M) & Sel(\langle M \leftarrow\!* n = N\rangle, m, P) & \mapsto_{\lambda\mathcal{O}bj} Sel(M, m, P) \end{array}$$

The main operation on objects is method invocation, whose reduction is defined by the second rule. Sending a message m to an object M, containing a method m, reduces to $Sel(M, m, M)$. More generally, in the expression $Sel(M, m, N)$, the term N represents the receiver (or recipient) of the message, the constant m is the message we want to send to the receiver of the message, and the term M is (or reduces to) a proper sub-object of N.

By looking at the last two rewrite rules, one may note that the Sel function "scans" the recipient of the message until it finds the definition of the method we want to use; when it finds the body of the method, it applies this body to the recipient of the message. The operational semantics in [14] was based on a more elaborate *bookkeeping* relation which transforms the receiver (*i.e.* an ordered list of methods) into another equivalent object where the method we are calling is always the last overridden one.

As a simple example of the calculus, we show an object which has the capability to extend itself simply by receiving a message which encodes the method to be added.

Example 6 (An object with "self-extension"). Consider the object *Self_ext* [17] $\langle\langle\,\rangle \leftarrow\!\!+\, add\_n = \lambda S.\langle S \leftarrow\!\!+\, n = \lambda S'.1\rangle\rangle$. If we send the message *add_n* to *Self_ext*, then we get $Self\_ext \Leftarrow add\_n \mapsto_{\lambda\mathcal{O}bj} Sel(Self\_ext, add\_n, Self\_ext) \mapsto_{\lambda\mathcal{O}bj}$ $(\lambda S.\langle S \leftarrow\!\!+\, n = \lambda S'.1\rangle)Self\_ext \mapsto_{\lambda\mathcal{O}bj} \langle Self\_ext \;\leftarrow\!\!+\, n = \lambda S'.1\rangle$, resulting in the method n being added to *Self_ext*.

The Translation of $\lambda\mathcal{O}bj$ into ρCal. The translation of a $\lambda\mathcal{O}bj$-term into a corresponding ρCal-term is quite trivial and can be done in the theory $\mathbb{T}_{\lambda\mathcal{O}bj}$ where the symbol "," is associative and *null* is its neutral element. Intuitively, an object in $\lambda\mathcal{O}bj$ is translated into a simple structure in ρCal. The choice we made for object override is an imperative one, *i.e.* we *delete* the method we are overriding using the *kill* function defined in Example 4. The translation is defined as follows:

$$\begin{array}{lll@{\qquad}lll}
[\![c]\!] & \triangleq & c & [\![\langle\,\rangle]\!] & \triangleq & null \\
[\![X]\!] & \triangleq & X & [\![\langle M \leftarrow n = N\rangle]\!] & \triangleq & kill_n([\![M]\!]), n \to [\![N]\!] \\
[\![\lambda X.M]\!] & \triangleq & X \to [\![M]\!] & [\![\langle M \leftarrow\!\!+ n = N\rangle]\!] & \triangleq & [\![M]\!], n \to [\![N]\!] \\
[\![MN]\!] & \triangleq & [\![M]\!]\bullet[\![N]\!] & [\![M \Leftarrow m]\!] & \triangleq & [\![M]\!].m \triangleq [\![M]\!]\bullet m\bullet[\![M]\!] \\
 & & & [\![Sel(M,m,N)]\!] & \triangleq & [\![M]\!]\bullet m\bullet[\![N]\!]
\end{array}$$

For instance, Example 7 shows an example of a simple computation in $\lambda\mathcal{O}bj$ and the corresponding translation into the ρCal, and Example 8 presents the translation of the *Self_ext* object into the ρCal.

Example 7 (A Simple Computation). Let *Point* be the simple diagonal point $\langle\langle\langle\,\rangle \leftarrow\!\!+ x = \lambda S.S \Leftarrow y\rangle \leftarrow\!\!+ y = \lambda S.1\rangle$. Then $Point \Leftarrow x \mapsto_{\lambda\mathcal{O}bj} Sel(Point, x, Point) \mapsto_{\lambda\mathcal{O}bj} Sel(\langle\langle\,\rangle \leftarrow\!\!+ x = \lambda S.S \Leftarrow y\rangle, x, Point) \mapsto_{\lambda\mathcal{O}bj} (\lambda S.S \Leftarrow y)Point \mapsto_{\lambda\mathcal{O}bj} Point \Leftarrow y \mapsto_{\lambda\mathcal{O}bj} Sel(Point, y, Point) \mapsto_{\lambda\mathcal{O}bj} (\lambda S.1)Point \mapsto_{\lambda\mathcal{O}bj} 1$.

The above computation in $\lambda\mathcal{O}bj$ can be easily translated into a corresponding computation in ρCal using $t \triangleq [\![Point]\!] \triangleq x \to S \to S.y, y \to S \to 1$ as follows: $[\![Point \Leftarrow x]\!] \triangleq t.x \triangleq (x \to S \to S.y, y \to S \to 1)\bullet x\bullet t \mapsto\!\!\!\to (S \to S.y, null)\bullet t =_{\mathbb{T}_{\lambda\mathcal{O}bj}} (S \to S.y)\bullet t \mapsto t.y \triangleq (x \to S \to S.y,\ y \to S \to 1)\bullet y\bullet t \mapsto\!\!\!\to (null, S \to 1)\bullet t =_{\mathbb{T}_{\lambda\mathcal{O}bj}} (S \to 1)\bullet t \mapsto 1$.

Example 8 (Translation of Self_ext). The object *Self_ext* can be easily translated in the ρCal as $t_1 \triangleq [\![Self\_ext]\!] \triangleq add\_n \to S \to (S, n \to S' \to 1)$. Then: $(t_1.add\_n).n \mapsto ((S \to (S, n \to S' \to 1))\bullet t_1).n \mapsto (t_1, n \to S' \to 1).n \triangleq \underbrace{(add\_n \to \ldots, n \to S' \to 1)}_{t_2}.n \mapsto\!\!\!\to null, (S' \to 1)\bullet t_2 =_{\mathbb{T}_{\lambda\mathcal{O}bj}} (S' \to 1)\bullet t_2 \mapsto 1$.

The translation into the ρCal can be proved correct when the theory $\mathbb{T}_{\lambda\mathcal{O}bj}$ is considered:

Theorem 2 (Translation of $\lambda\mathcal{O}bj$ into ρCal). *If $M \mapsto_{\lambda\mathcal{O}bj} N$, then $[\![M]\!] \mapsto\!\!\!\to_{\mathbb{T}_{\lambda\mathcal{O}bj}} [\![N]\!]$.*

4.2 The Object Calculus

The Object Calculus [1] is a calculus where the only existing entities are the objects; it is computationally complete since λ-calculus, fixed points and complex structures can be easily encoded within it. A large collection of variants (functional and imperative, typed and untyped) for this calculus are presented in the book and in the literature.

Syntax and Operational Semantics. The syntax of the object calculus is defined as follows:

$$a, b ::= X \mid [m_i = \varsigma(X)b_i]^{i=1\ldots n} \mid a.m \mid a.m := \varsigma(X)b$$

Let $a \triangleq [m_i = \varsigma(X)b_i]^{i=1\ldots n}$; the small-step semantics is:

$$\begin{array}{lll} a.m_j & \mapsto_{\varsigma\mathit{Obj}} [X/a]b_j & j = 1\ldots n \\ a.m_j := \varsigma(X)b & \mapsto_{\varsigma\mathit{Obj}} [m_i = \varsigma(X)b_i, m_j = \varsigma(X)b]^{i=1\ldots n\setminus\{j\}} & j = 1\ldots n \end{array}$$

The Translation into ρCal. The translation of an $\varsigma\mathit{Obj}$-term into a corresponding ρCal-term is quite similar to the one of $\lambda\mathit{Obj}$, and can be done in the theory $\mathbb{T}_{\varsigma\mathit{Obj}}$ where the symbol "," is associative and commutative, and *null* is its neutral element. Given the function $kill_m \triangleq (X, m \to Y) \to X$, and the alias $(t_1.m := t_2) \triangleq (kill_m(t_1), m \to t_2)$, the translation is defined as follows:

$$\begin{array}{llllll} [\![X]\!] & \triangleq & X & [\![\, [m_i = \varsigma(X)b_i]^{i=1\ldots n} \,]\!] & \triangleq & (m_i \to X \to [\![b_i]\!])^{i=1\ldots n} \\ [\![a.m_j]\!] & \triangleq & [\![a]\!].m_j & [\![a.m := \varsigma(X)b]\!] & \triangleq & [\![a]\!].m := X \to [\![b]\!] \end{array}$$

As a simple example, we present the usual Abadi and Cardelli's encoding of the Point class [1].

Example 9 (A Point Class). The object *PClass* is defined in $\varsigma\mathit{Obj}$ as follows:

$$[new = \varsigma(S)o, val = \lambda S'.\ v(1\ 1), get = \lambda S'.\ (S'.val), set = \lambda S'.\lambda N.\ S'.val := N]$$

with $o \triangleq [val = \varsigma(S')(S.val)(S'), get = \varsigma(S')(S.get)(S'), set = \varsigma(S')(S.set)(S')]$, and it is translated into the ρCal as follows:

$$\begin{array}{l} new \to S \to t, val \to S \to S' \to v(1\ 1), \\ get \to S \to S' \to S'.val, set \to S \to S' \to v(X\ Y) \to (S'.val := S'' \to v(X\ Y)) \\ \text{with } t \triangleq (val \to S' \to (S.val)\bullet S', get \to S' \to (S.get)\bullet S', set \to S' \to (S.set)\bullet S') \end{array}$$

It is not hard to verify that $[\![PClass]\!].new \mapsto\!\!\!\!\to_{\mathbb{T}_{\varsigma\mathit{Obj}}} Point_{imp}$.

As another example, we present the Abadi and Cardelli's fixed point object operator. To do this we recall the usual encoding of lambda calculus in $\varsigma\mathit{Obj}$:

$$\begin{array}{lll} [\![S]\!] & \triangleq & S \\ [\![M\,N]\!] & \triangleq & [\![M]\!] \circ [\![N]\!] \end{array} \quad [\![\lambda S.M]\!] \triangleq [arg = \varsigma(S)S.arg, val = \varsigma(S)[S.arg/S][\![M]\!]]$$

and the alias $p \circ q \triangleq (p.arg := \varsigma(S)q).val$, which represents the encoding of the function application.

Example 10 (Another Fixed-Point Object). In $\varsigma\mathit{Obj}$, the generic fixed-point object $Fix \triangleq [arg = \varsigma(S)S.arg, val = \varsigma(S)((S.arg).arg := \varsigma(S')S.val).val]$, can be translated into ρCal as:

$$Fix \triangleq arg \to S \to S.arg, val \to S \to (kill_{arg}(S.arg), arg \to S' \to S.val).val$$

Using the aliases $t_1 \circ t_2 \triangleq (t_1.arg := S \to t_2).val$, and $Fix_f \triangleq Fix.arg := S \to f$, we can prove that $Fix \circ f \equiv Fix_f.val \mapsto\!\!\!\!\to ((Fix_f.arg).arg := S' \to Fix_f.val).val \mapsto\!\!\!\!\to (f.arg := S' \to Fix \circ f).val \equiv f \circ (Fix \circ f)$.

The translation into the ρCal can be proved correct when the theory $\mathbb{T}_{\varsigma Obj}$ is considered:

Theorem 3 (Translation of ςObj into ρCal). *If $M \mapsto_{\varsigma Obj} N$, then $[\![M]\!] \mapsto\!\!\!\!\rightarrow_{\mathbb{T}_{\varsigma Obj}} [\![N]\!]$.*

The following example shows that the expressivity of ρCal is strictly stronger than the two previous calculi of objects as they cannot be translated neither in λObj nor in ςObj. In fact, we can easily consider "labels" and "bodies" as *first-class entities* that can be passed as function arguments.

Example 11 (The Daemon and the Para object).

1. Assume $Para$ be $a \to S \to b, par(X) \to S \to S.X$. This object has a method $par(X)$ which seeks for a method name that is assigned to the variable X and then sends this method to the object itself. Then: $Para.(par(a)) \triangleq Para\bullet(par(a))\bullet Para \mapsto\!\!\!\!\rightarrow (S \to S.a)\bullet Para \mapsto Para.a \mapsto\!\!\!\!\rightarrow b$.
2. Assume $Daemon$ be $set \to S \to X \to (X, set \to S' \to Y \to (Y, S'))$. The set method of $Daemon$ is used to create an object completely from scratch by receiving from outside all the components of a method, namely, the labels and the bodies. Once the object is installed, it has the capability to extend itself upon the reception of the same message set. In some sense the "power" of $Daemon$ has been inherited by the created object. Then: $Daemon.set(x \to S \to 3) \triangleq Daemon\bullet set\bullet Daemon\bullet(x \to S \to 3) \mapsto\!\!\!\!\rightarrow (X \to (X, set \to S' \to Y \to (Y, S')))\bullet(x \to S \to 3) \mapsto x \to S \to 3, set \to S' \to Y \to (Y, S') \triangleq t$, and $t.set(y \to S \to 4) \mapsto\!\!\!\!\rightarrow y \to S \to 4, t$.

One may wonder if some reductions in the ρCal can be translated into a suitable computation either in the λObj calculus or in the ςObj calculus: we can distinguish the following cases:

– reductions *à la* lambda calculus, like $(X \to t_1)\bullet t_2 \mapsto [X/t_2]t_1$, *i.e.* with trivial matching, are directly translated in either λObj and ςObj;
– reductions which use non-trivial matching, like $(a \to b)\bullet c \mapsto null$, can be translated in either λObj and ςObj, modulo an encoding of the underlined matching theory (taking into account also possible failures), as in $Sol(a \ll_{\mathbb{T}} c) = null$;
– reductions which use structures, like $(X \to a, X \to b)\bullet c \mapsto (a, b)$, can be translated in either λObj and ςObj, modulo a non trivial encoding of the nondeterministic features intrinsic to the ρCal.

Therefore, when using more elaborate theories and structures, the encoding becomes at least as difficult as the matching problem underlining the theory itself.

5 Conclusions and Further Work

We have presented a new version of the Rho Calculus and shown that its embedded matching power permits us to uniformly and naturally encode various calculi including the Lambda Calculus of Objects and the Object Calculus.

This presentation of the Rho Calculus inherits from the ideas and concepts of the first proposed one [6,7], it simplifies the rules of the calculus and improves the way the results are handled. This allows us first to encode object oriented calculi in a very natural and simple way but further to design new powerful object oriented features, like parameterized methods or self creating objects. Based on this new generic approach, an implementation of objects is under way in the Rho-based language ELAN [13]. More generally, rewrite based languages like ASF+SDF, CafeOBJ, Maude, Stratego, or ELAN, could benefit from a Rho-based semantics that gives a first-class status to rewrite rules and to their application.

We are now planning to work on several directions. First, on giving a big-step semantics in order to define a deterministic evaluation strategy, when needed. Then, the calculus could be further generalized by the explicit use of constraints. For the moment, the (ρ) rule calls for the solutions set of the relevant matching constraint; this could be replaced by an appropriate constrained term, in the spirit of constraint programming. We are also exploring an elaborated type system allowing in particular to type self-applications. As we have seen, the are many possible applications of the framework; a track that we have not yet mention in this paper concerns encoding concurrency in the spirit of the early work of Viry [37].

Independently of these ongoing works, we believe that the matching power of the Rho Calculus could be widely used, thanks to its expressiveness and simplicity, as a new model of computation.

Acknowledgement. We thank the referees for their constructive remarks, Hubert Dubois and all the members of the ELAN group for their comments and interactions on the topics of the Rho Calculus.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer Verlag, 1996.
2. H. Ait-Kaci, A. Podelski, and G. Smolka. A feature constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1-2):263–283, 1994.
3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In *Proc. of WRLA*, volume 15. Electronic Notes in Theoretical Computer Science, 1998.
4. V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. of LICS*, pages 82–90, 1988.
5. H.-J. Bürckert. Matching — A special case of unification? *Journal of Symbolic Computation*, 8(5):523–536, 1989.
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. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, 2001.
8. H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. Technical Report A00-R-363, LORIA, Nancy, 2000.

9. H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180. Springer-Verlag, 2001.
10. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proc. of WRLA*, volume 4. Electronic Notes in Theoretical Computer Science, 1996.
11. L. Colson. Une structure de données pour le λ-calcul typé. Private Communication, 1988.
12. G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.
13. Hubert Dubois and Hélène Kirchner. Objects, rules and strategies in `ELAN`. , 2000. Submitted.
14. K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
15. K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of FEM*, 1997.
16. J. Gallier and V. Breazu-Tannen. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *Proc. of ICALP*, volume 372 of *LNCS*, pages 137–150. Springer-Verlag, 1989.
17. P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Proc. of OOPSLA*, pages 166–178. The ACM Press, 1998.
18. J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In *Proc. of CTRS*, volume 308 of *LNCS*, pages 258–263. Springer-Verlag, 1987.
19. C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.
20. G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
21. J.-M. Hullot. Associative-commutative pattern matching. In *Proc. of IJCAI*, 1979.
22. J.P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 1997.
23. C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.
24. C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.
25. J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.
26. Donald E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.
27. A. Laville. Lazy pattern matching in the ML language. In *Proc. FCT & TCS*, volume 287 of *LNCS*, pages 400–419. Springer-Verlag, 1987.
28. D Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proc. of ELP*, volume 475 of *LNCS*, pages 253–281. Springer-Verlag, 1991.
29. Tobias Nipkow and Christian 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.

30. M. Okada. Strong normalizability for the combined system of the typed λ calculus and an arbitrary convergent term rewrite system. In *Proc. of ISSAC*, pages 357–363. ACM Press, 1989.
31. V. Padovani. Decidability of fourth-order matching. *Mathematical Structures in Computer Science*, 3(10):361–372, 2000.
32. G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.
33. S. Peyton-Jones. *The implementation of functional programming languages.* Prentice Hall, Inc., 1987.
34. Équipe Protheo. The Elan Home Page, 2001. `http://elan.loria.fr`.
35. A. van Deursen, J. Heering, and P. Klint. *Language Prototyping.* World Scientific, 1996.
36. V. van Oostrom. Lambda calculus with patterns. Technical Report IR-228, Vrije Universiteit, November 1990.
37. P. Viry. Input/Output for ELAN. In *Proc. of WRLA*, volume 4. Electronic Notes in Theoretical Computer Science, 1996.
38. E. Visser and Z.e.A. Benaissa. A core language for rewriting. In *Proc. of WRLA*, volume 15. Electronic Notes in Theoretical Computer Science, 1998.
39. D. A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993.