Matching Power

Horatiu Cirstea, Claude Kirchner, Luigi Liquori

Matching Power

–Extended Abstract–

Horatiu Cirstea[1], Claude Kirchner[2], and Luigi Liquori[3]

[1] LORIA and UHP 54506 Vandoeuvre-lès-Nancy BP 239 Cedex France
Horatiu.Cirstea@loria.fr
[2] LORIA and INRIA 54506 Vandoeuvre-lès-Nancy BP 239 Cedex France
Claude.Kirchner@loria.fr
[3] LORIA and INPL-ENSMN 54042 Parc de Saurupt
Luigi.Liquori@loria.fr

Abstract. In this paper we give a new simpler and uniform presentation of the rewriting calculus also called *Rho Calculus*. In addition to its simplicity, this reformulation explicitly allows us to encode complex structures such as lists, sets, and objects. We provide extensive examples of calculus use and we focus on its properties and 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. This enlightens the capabilities of the rewriting calculus based language ELAN to be used as a logical as well as powerful semantical framework. *In summa*, we intend to show that the Rho Calculus represents a *lingua franca* to encode many paradigms of computations.

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 [KMP77], trees [HO82] or feature objects [AKPS94].

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 [Mil96,Lav87]. It is somewhat astonishing that one of the most common 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 [PJ87,vO90] or by the introduction of matching and rewrite rules in functional programming languages like ML. 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 [KvOvR93] and other higher-order rewriting systems [Wol93,NP98], in the second case the works on combination of lambda calculus with term rewriting [Oka89,BT88,GBT89,JO97] to mention only a few.

Embedding more information in the matching process makes it appropriate to deal with complex tasks like program transformations [HL78] or theorem proving [PS81]. In that direction, matching in elaborated theories has been also studied extensively, either in equational theories [Hul79,Bür89] or in higher-order logic [Dow94,Pad96] 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 rule

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 rule application results explicit, the Rho Calculus has the ability to handle non-determinism in the sense of sets of results: an empty set of results represents an application failure, a singleton represents a deterministic result and a set with more than one element represents a non-deterministic choice between the elements of the set.

Rewriting is central in several programming languages developed since the seventies. Amongst the main ones let us mention OBJ [GKK+87], ASF+SDF [DHK96], Maude [CELM96], CafeOBJ [FN97] and ELAN [KKV95,BKK+98][1] which has been at the origin of some of the main concepts of the rewriting calculus. In return, the Rho Calculus provides a natural semantics to such languages and in particular to ELAN, covering the notion of rule application strategy, an important concept of the language.

The Rho Calculus offers a broad spectrum of applications due to the two fundamental parameters of the calculus: the theory modulo which matching is performed and the structure under which the results of a rule application are returned. Adjusting these parameters to various situations permits us to easily describe in a uniform but still appropriately tuned manner main calculi, namely: lambda calculus, term rewriting and object calculi.

The contributions of this paper are therefore the following.

- First a description of a new version of the Rho Calculus introduced in [CK99a,CK99b,Cir00] is given. We provide here 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;
- Second, 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;
- Third, 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 [AC96] and the *Lambda Calculus of Objects* of Fisher, Honsell, and Mitchell [FHM94] ($\lambda\mathcal{O}bj$). Moreover we show two examples in Rho Calculus that cannot be encoded in the above calculi;
- Finally, we present properties of the Rho Calculus (in Appendix A).

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 the Object Calculus in the Rho Calculus. Conclusions and further works are finally discussed in Section 5. In Appendix A we state some properties of the Rho Calculus.

2 Syntax and Semantics

Notational Conventions. In this paper, the symbol t ranges over the set $\mathcal{T}$ of terms, the symbols $S, X, Y, Z, \ldots$ range over the infinite set $\mathcal{V}$ of variables, the symbols $a, b, \ldots, z, 0, 1, 2, 3, \ldots, null, \oplus, \circ, \ldots$ range over the infinite set $\mathcal{C}$ of constants of fixed arity. All symbols can be indexed. The symbol $\equiv$ denotes syntactic identity of terms. We work modulo α-conversion, denoted $\alpha_{\mathcal{X}}$ and defined in Appendix A.1. We also follow the Barendregt convention [Bar84], saying that free and bound variables have different names.

2.1 Syntax

The syntax of the ρCal is defined as follows:

$$t ::= a \mid X \mid t \to t \mid t \bullet t \mid \qquad \text{plain terms}$$

$$null \mid t, t \qquad \text{structured terms}$$

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

[1] For details, see `http://elan.loria.fr`.

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 are obtained. We will come back to this later but to support the intuition let us mention that if we define the "," operator to be associative, then a list structure is obtained and if the "," operator is specified as associative, commutative and idempotent then a set structure is obtained. The expression *null* denotes an empty structure.

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

Definition 1 (Some Type Signatures). *The type-signature of the special constant symbol null and of the* $\rightarrow$, "$\bullet$" *and* "," *operators is as follows:*

$$\begin{aligned} null &: \mathcal{T} \\ \rightarrow &: \mathcal{T} \times \mathcal{T} \Rightarrow \mathcal{T} \\ \bullet &: \mathcal{T} \times \mathcal{T} \Rightarrow \mathcal{T} \\ , &: \mathcal{T} \times \mathcal{T} \Rightarrow \mathcal{T} \end{aligned}$$

Definition 2 (Abbreviations). *In the paper we adopt the following abbreviations:*

$$\begin{array}{lcll} t_1.t_2 & \triangleq & t_1 \bullet t_2 \bullet t_1 & \textit{self-application} \\ t(t_1 \ldots t_n) & \triangleq & t \bullet t_1 \ldots \bullet t_n & \textit{function-application} \\ (t_i)^{i=1\ldots n} & \triangleq & t_1, \ldots, t_n & \textit{structure} \end{array}$$

where $n \in \mathsf{N}^*$.

We draw the attention of the reader on the main difference between "$\bullet$" denoting in this paper *application* and "." denoting the object-oriented *self-application* operator. To denote iteration in tuples, we will always use "...", whatever operator is iterated.

Example 1 (ρCal-terms). We list some simple examples of ρCal-terms:

- The term $(f(X) \rightarrow X) \bullet f(a)$ represents the application of the classical rewrite rule $f(X) \rightarrow X$ to the term $f(a)$;
- The term $X \rightarrow Y \rightarrow X$ corresponds to the lambda-term $\lambda(X)\lambda(Y)X$;
- The term $(1, 2, 3)$ denotes a simple ternary structure;
- The term $(cx \rightarrow 0, cy \rightarrow 0)$ denotes a record with two fields, cx and cy.

2.2 Matching Theories

An important parameter of the ρCal is the matching theory $\mathbb{T}$. We assume given a theory $\mathbb{T}$, defined for example equationally, and we define matching problems in this general setting. In what follows we introduce several such theories, some of them being used later for instantiating the general ρCal.

Definition 3 (Matching theories). *We list some useful theories:*

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

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\ (Trans) \qquad \frac{t_1 = t_2}{t_2 = t_1}\ (Symm)$$

$$\frac{t_1 = t_2}{t_3[t_1]_p = t_3[t_2]_p}\ (Ctx) \qquad \frac{\mathcal{X} \subseteq \mathcal{V}}{t = \alpha_{\mathcal{X}}(t)}\ (Refl_\alpha)^2$$

where $t_1[t_2]_p$ *denotes the term* t_1 *with the term* t_2 *at position* p.

[2] See Appendix A.1 for a formal definition of the α-conversion $\alpha_{\mathcal{X}}(t)$.

– *For a given binary symbol f, the theory of Commutativity $\mathbb{T}_{C(f)}$ is defined by $\mathbb{T}_\emptyset$ plus the following inference rule:*

$$\frac{}{f(t_1\ t_2) = f(t_2\ t_1)}\quad (Comm)$$

– *For a given binary symbol f, the theory of Associativity $\mathbb{T}_{A(f)}$ is defined by $\mathbb{T}_\emptyset$ plus the following inference rule:*

$$\frac{}{f(f(t_1\ t_2)\ t_3) = f(t_1\ f(t_2\ t_3))}\quad (Assoc)$$

– *For a given binary symbol f, the theory of Idempotency $\mathbb{T}_{I(f)}$ is defined by $\mathbb{T}_\emptyset$ plus the following inference rule:*

$$\frac{}{f(t\ t) = t}\quad (Idem)$$

– *For a given binary symbol f and a constant 0, the theory of Neutral (Right and Left) Element $\mathbb{T}_{N(f^0)}$ is defined by $\mathbb{T}_\emptyset$ plus the following inference rules:*

$$\frac{}{f(t\ 0) = t}\quad (Zero_R) \qquad \frac{}{f(0\ t) = t}\quad (Zero_L)$$

– *The MultiSet theory, $\mathbb{T}_{MSet(f,nil)}$, is obtained by considering the symbol f as an associative and commutative function symbol with nil as a neutral element, i.e.:*

$$\mathbb{T}_{MSet(f,nil)} = \mathbb{T}_{A(f)} \cup \mathbb{T}_{C(f)} \cup \mathbb{T}_{N(f^{nil})}$$

In what follows, an important case is when the "," and null operators of the calculus verify the multiset axioms, in this case $\mathbb{T}_{MSet(\text{“,”},null)}$ is simply denoted $\mathbb{T}_{MSet}$.

– *The Set theory, $\mathbb{T}_{Set(f,nil)}$, is obtained from $\mathbb{T}_{MSet(f,nil)}$, by considering the symbol f as an idempotent function symbol, i.e.:*

$$\mathbb{T}_{Set(f,nil)} = \mathbb{T}_{MSet(f,nil)} \cup \mathbb{T}_{I(f)}$$

As for multisets, $\mathbb{T}_{Set(\text{“,”},null)}$ is simply denoted $\mathbb{T}_{Set}$.

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

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

Note that in the theory $\mathbb{T}_\emptyset$ we do not need an inference rule describing the stability by substitution since all the above theories are defined using inference rules exhibiting all possible instantiations for the terms.

As already mentioned, computing the substitutions which solve the matching (in the theory $\mathbb{T}$) from a pattern t_1 to a subject t_2 is a fundamental element of the ρCal.

Definition 4 (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 $\mathcal{S}$ as returning the $\prec$-ordered [3] list of all T-matches of $\mathcal{S}$ when $\mathcal{S}$ is not trivial and the list containing only $\sigma_{\mathbf{id}}$, where $\sigma_{\mathbf{id}}$ is the identity substitution, when $\mathcal{S}$ is trivial.*

Notice that when the matching algorithm fails (*i.e.* returns $\mathbb{F}$), the function $\mathcal{S}ol$ returns the empty list.

Since in general we can consider arbitrary theories over ρCal-terms, $\mathbb{T}$-matching is in general undecidable, even when restricted to first-order equational theories [JK91]. Among the decidable cases we can mention higher-order-pattern matching that is decidable and unitary as a consequence of the decidability of pattern unification [Mil91,DHKP96], higher-order matching which is known to be decidable up to the fourth order [Pad96,Dow94,HL78] (the decidability of the general case being still open), and many first-order equational theories including associativity, commutativity, distributivity and most of their combinations [Nip89,Rin96]. The syntactic matching substitution between two first-order terms, when it exists, is unique and can be computed by a simple recursive algorithm given for example by G. Huet [Hue76].

In $\mathbb{T}_\emptyset$, the matching substitution from a ρCal-term t_1 to a ρCal-term t_2 can be computed by the following rewrite system, 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 "$\bullet$" or "$\rightarrow$".

$$(Decompose) \quad \diamond_1(t_1 \dots t_n) \ll_{\mathbb{T}_\emptyset} \diamond_2(t'_1 \dots t'_m) \rightsquigarrow \begin{cases} \bigwedge_{i=1\dots n}(t_i \ll_{\mathbb{T}_\emptyset} t'_i) & \text{if } \diamond_1 \equiv \diamond_2 \text{ and } n = m \\ \mathbb{F} & \text{otherwise} \end{cases}$$

$$(Merge) \quad (X \ll_{\mathbb{T}_\emptyset} t) \wedge (X \ll_{\mathbb{T}_\emptyset} t') \rightsquigarrow \begin{cases} (X \ll_{\mathbb{T}_\emptyset} t) & \text{if } t =_{\mathbb{T}_\emptyset} t' \\ \mathbb{F} & \text{otherwise} \end{cases}$$

$$(Clash) \quad t \ll_{\mathbb{T}_\emptyset} X \rightsquigarrow \mathbb{F} \quad \text{if } t \notin \mathcal{V}$$

$$(Propagate) \quad \mathbb{F} \wedge (t \ll_{\mathbb{T}_\emptyset} t') \rightsquigarrow \mathbb{F}$$

Starting from a matching system $\mathcal{S}$, the application of this rule set terminates and returns either $\mathbb{F}$ when there are no substitutions solving the system, or a system $\mathcal{S}'$ in "normal form" from which the solution can be trivially inferred [KK99].

This set of rules could be easily extended to matching modulo commutativity (but the number of matches can then be exponential in the size of the initial problem). It is also decidable for associativity-commutativity [Hul79] but in general it could be as difficult as unification [Bür89].

Example 2 (Matching).

1. In $\mathbb{T}_\emptyset$, the equation $a \ll_{\mathbb{T}_\emptyset} b$ has no solution, and thus $\mathcal{S}ol(a \ll_{\mathbb{T}_\emptyset} b)$ returns the empty list of substitutions;
2. In $\mathbb{T}_\emptyset$, $a \ll_{\mathbb{T}_\emptyset} a$ is solved by all substitutions and thus $\mathcal{S}ol(a \ll_{\mathbb{T}_\emptyset} a)$ returns the list with only one element $\sigma_{\mathbf{id}}$;
3. In $\mathbb{T}_\emptyset$, the equation $f(X\ g(X\ Y)) \ll_{\mathbb{T}_\emptyset} f(a\ g(a\ b))$ has as solution the substitution $\sigma \equiv [X/a\ Y/b]$;
4. In $\mathbb{T}_{C(f)}$ (*i.e.* f is a commutative symbol) the equation $f(X\ Y) \ll_{\mathbb{T}_{C(f)}} f(a\ b)$ has the two solutions $\sigma_1 \equiv [X/a\ Y/b]$ and $\sigma_2 \equiv [X/b\ Y/a]$ and therefore the matching equation $\mathcal{S}ol(f(X\ Y) \ll_{\mathbb{T}_{C(f)}} f(a\ b))$ is solved by the list $\sigma_1\ \sigma_2$.
5. In $\mathbb{T}_{AN} = \mathbb{T}_{A(,)} \cup \mathbb{T}_{N(,^{null})}$, the equation $(X, a, Y) \ll_{\mathbb{T}_{AN}} (null, b, a, c, d)$ has as solution the substitution $[X/(null, b)\ Y/(c, d)]$, while the two equations $(X, a, Y) \ll_{\mathbb{T}_{AN}} (null, a)$ and $(X, a, Y) \ll_{\mathbb{T}_{AN}} a$ have both the same solution $[X/null\ Y/null]$, since $a =_{\mathbb{T}_{AN}} (null, a) =_{\mathbb{T}_{AN}} (null, a, null)$;
6. In $\mathbb{T}_{ACN} = \mathbb{T}_{A(,)} \cup \mathbb{T}_{C(,)} \cup \mathbb{T}_{N(,^{null})}$, the equation $(X, a) \ll_{\mathbb{T}_{ACN}} (null, b, a, c, d)$ has as solution the substitution $[X/null, b, c, d]$ since $(null, b, a, c, d) =_{\mathbb{T}_{ACN}} (null, b, c, d, a)$, while the two matching equations $(X, a) \ll_{\mathbb{T}_{ACN}} (null, a)$ and $(X, a, Y) \ll_{\mathbb{T}_{AN}} a$ have both the same solution $[X/null\ Y/null]$.

[3] We consider a total order $\prec$ on the set of substitutions (see Example 15 in Appendix A.1). Such orderings always exist: one can for example take the lexicographic ordering on the flattened representation of the substitutions. The ordering on substitutions can be seen as a parameter of the ρCal and thus can be customized by the user.

2.3 Operational Semantics

For a given ordering $\prec$ (which is left implicit in the notation) and a theory $\mathbb{T}$, the operational semantics is defined by the computational rules given in Figure 1.

$$(Fire-\rho) \quad (t_1 \to t_2)\bullet t_3 \quad \mapsto_{\mathbb{T}} \quad \begin{cases} null & \text{when } t_1 \ll_{\mathbb{T}} t_3 \text{ has no solution} \\ \sigma_1 t_2, \ldots, \sigma_n t_2 & \text{where } \sigma_i \in Sol(t_1 \ll_{\mathbb{T}} t_3) \ \ (n \leq \infty) \end{cases}$$

$$(Distrib-\epsilon) \quad (t_1, t_2)\bullet t_3 \quad \mapsto_{\mathbb{T}} \quad t_1\bullet t_3, t_2\bullet t_3$$

$$(Distrib'-\nu) \quad null\bullet t \quad \mapsto_{\mathbb{T}} \quad null$$

Fig. 1. Evaluation rules of the ρCal.

The central idea of the main rule of the calculus ($Fire$ also abbreviated ρ) is that the application of a rewrite rule $t_1 \to t_2$ at the root (also called top) 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 $Sol(t_1 \ll_{\mathbb{T}} t_3)$ to the term t_2. When there 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 there could be an infinity of solutions to the matching problem $(t_1 \ll_{\mathbb{T}} t_3)$ in some theories [FH86], but in this paper we restrict ourselves to the case where the matching problem is finitary. Possible ways to deal with the infinitary case are described in [Cir00]. It is important to remark that if t_1 is a variable, then the $Fire$-rule corresponds exactly to the β-rule of the lambda calculus.

The $Distrib$-rules, abbreviated (ϵ) and (ν), deal with the distributivity of the application on the structures whose constructors are "," and $null$.

With respect to the previous presentation of the Rho Calculus [CK99a,CK99b,Cir00], 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.

When the theory $\mathbb{T}$ is clear from the context, its denotation will be omitted.

When working modulo reasonably powerful theories $\mathbb{T}$, the evaluation rules of the ρCal are confluent, for example:

Theorem 1 (Confluence in $\mathbb{T}_\emptyset$). *Given a term t_1 such that all its rewrite rules contain no abstraction in the left-hand side, if $t_1 \mapsto\!\!\!\!\to t_2$ and $t_1 \mapsto\!\!\!\!\to t_3$ then there exists a term t_4 such that $t_2 \mapsto\!\!\!\!\to t_4$ and $t_3 \mapsto\!\!\!\!\to t_4$.*

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 3 (In the Basic Theory $\mathbb{T}_\emptyset$).

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 $\sigma_{\mathbf{id}}$ like in $(X \to X)\bullet a \overset{\rho}{\mapsto} [X/a]X \equiv a$;
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. Function composition can be easily represented 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* of [PJ87] can be easily represented in the ρCal. For instance, the lambda-term $\lambda Pair(X\,Y).X$, which selects the first element of a pair and its application to the term $Pair(a\ b)$ can be represented and reduced as follows: $(Pair(X\ Y) \to X)\bullet Pair(a\ b) \overset{\rho}{\mapsto} [X/a]X \equiv a$;
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

$$Y_\lambda \triangleq (A_\lambda\ A_\lambda) \text{ with } A_\lambda \triangleq \lambda(X)\lambda(Y)Y(XXY)$$

which can be translated in the ρCal as follows:

$$Y_\rho \triangleq A_\rho\bullet A_\rho \text{ with } A_\rho \triangleq X \to Y \to Y\bullet(X\bullet X\bullet Y)$$

Then, we have:

$$\begin{aligned} Y_\rho\bullet t &\triangleq A_\rho\bullet A_\rho\bullet t \\ &\triangleq (X \to Y \to Y\bullet(X\bullet X\bullet Y))\bullet A_\rho\bullet t \\ &\overset{\rho}{\mapsto} (Y \to Y\bullet(A_\rho\bullet A_\rho\bullet Y))\bullet t \\ &\overset{\rho}{\mapsto} t\bullet(A_\rho\bullet A_\rho\bullet t) \\ &\triangleq t\bullet(Y_\rho\bullet t) \end{aligned}$$

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 ([Cir00]);
9. Consider the following three functions with the following signatures and behavior:

$$\begin{aligned} car &: \mathcal{T} \Rightarrow \mathcal{T} &&\triangleq X, Y \to X \\ cdr &: \mathcal{T} \Rightarrow \mathcal{T} &&\triangleq X, Y \to Y \\ cons &: \mathcal{T} \Rightarrow \mathcal{T} \Rightarrow \mathcal{T} &&\triangleq X \to Y \to (X, Y) \end{aligned}$$

It is easy to verify that $car(a, b, c, null) \mapsto\!\!\!\!\to a$, and that $cdr(a, b, c, null) \mapsto\!\!\!\!\to b, c, null$. For the $cons$ function we obtain $cons(d\ a, b, c, null) \mapsto\!\!\!\!\to d, a, b, c, null$.

Example 4 (In $\mathbb{T}_A$, $\mathbb{T}_C$, and $\mathbb{T}_{AC}$).

1. In $\mathbb{T}_{A(\circ)}$, consider the term $\circ(X\ Y) \to X$. When applying it to $\circ(a\ \circ(b\ \circ(c\ d)))$, then, thanks to the associativity of $\circ$, we obtain as result of the evaluation the term $(a, \circ(a\ b), \circ(a\ \circ(b\ c)))$;
2. In $\mathbb{T}_{C(\oplus)}$ let $\oplus(X\ Y) \to X$ be the rewrite rule that selects either X or Y. In $\mathbb{T}_C$ the application $(\oplus(X\ Y) \to X)\bullet \oplus (a\ b)$ evaluates to (a, b), a structure representing all possible results;
3. In $\mathbb{T}_{AC(\oplus)}$ 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 and commutativity of the $\oplus$ operator, while the elimination of doubles is performed by the rewrite rule;

Example 5 (In $\mathbb{T}_{N(f^0)}$, $\mathbb{T}_{MSet}$, and $\mathbb{T}_{Set}$).

1. In $\mathbb{T}_{N(f^0)}$. Using a theory with a neutral element allows us to "ignore" variables from the rewrite rules. For example, in $\mathbb{T}_{N(\oplus^0)}$ 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 $b \oplus a \oplus b$ obviously reduces to $b \oplus b \oplus b$ and the same rule applied to a leads to b since $a =_{\mathbb{T}_{N(\oplus^0)}} 0 \oplus a \oplus 0$;
2. In $\mathbb{T}_{MSet}$. In this case we can directly represent structures as sets in the calculus itself. For instance, the function that transforms a multiset into a set by eliminating doubles can be directly encoded as follows:

$$idem : \mathcal{T} \Rightarrow \mathcal{T} \triangleq (X, X, Y) \to (X, Y)$$

It is easy to verify that $idem(a, b, c, a, b) \mapsto (a, b, c)$. The theory plays a crucial role when we apply $idem$ to $(a, a) =_{\mathbb{T}_{MSet}} (a, a, null)$ and in this case we obtain the result a.

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 can be 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 6 (Examples 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, $t.\omega \overset{\rho}{\mapsto} (S \to S.\omega){\bullet}t \overset{\rho}{\mapsto} t.\omega \mapsto\!\!\!\!\to \ldots$;
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, $t.ping \triangleq t{\bullet}ping{\bullet}t \overset{\epsilon}{\mapsto} ((ping \to S \to S.pong){\bullet}ping, (pong \to S \to S.ping){\bullet}ping){\bullet}t \overset{\rho}{\mapsto}\!\!\!\!\to$ $(S \to S.pong){\bullet}t, null =_{\mathbb{T}_{\lambda\mathcal{O}bj}} (S \to S.pong){\bullet}t \overset{\rho}{\mapsto} t.pong \mapsto\!\!\!\!\to \ldots\ t.ping \mapsto\!\!\!\!\to \ldots$.

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 [FHM94];
- The *Object Calculus* of Abadi and Cardelli [AC96]

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 6, structured-terms are well suited to represent objects and to model the special "metavariable" `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 7 (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:

$$\begin{aligned} Point \triangleq\ & val \to S \to v(1\ 1), \\ & get \to S \to S.val, \\ & set \to S \to v(X\ Y) \to (S, val \to S' \to v(X\ Y)) \end{aligned}$$

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:

$$\begin{aligned} Point.get &\mapsto\!\!\!\!\rightarrow v(1\ 1) \\ Point.set(v(2\ 2)) &\mapsto\!\!\!\!\rightarrow Point, (val \rightarrow S' \rightarrow v(2\ 2)) \\ Point.set(v(2\ 2)).get &\mapsto\!\!\!\!\rightarrow v(1\ 1), v(2\ 2) \end{aligned}$$

Nota bene

- 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.* $val \rightarrow S' \rightarrow v(2\ 2)$;
- The call $Point.set(v(2\ 2)).get$ produces a structure composed of two elements, the former representing the value of the attribute val before the execution of set (hence before a side effect) and the latter one after the execution of set;
- 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 useful formalism to study side effects in imperative calculi;
- A way to fix imperative features is to modify the encoding of the method set by considering the term:

  $$kill_n : \mathcal{T} \Rightarrow \mathcal{T} \triangleq (X, n \rightarrow Z, Y) \rightarrow X, Y$$

  and defining the new object $Point_{imp}$ as follows:

  $$\begin{aligned} Point_{imp} \triangleq\ & val \rightarrow S \rightarrow v(1\ 1), \\ & get \rightarrow S \rightarrow S.val, \\ & set \rightarrow S \rightarrow v(X\ Y) \rightarrow (kill_{val}(S), val \rightarrow S' \rightarrow v(X\ Y)) \end{aligned}$$

  with the following behavior:

  $$\begin{aligned} Point_{imp}.get &\mapsto\!\!\!\!\rightarrow v(1\ 1) \\ Point_{imp}.set(v(2\ 2)) &\mapsto\!\!\!\!\rightarrow val \rightarrow S' \rightarrow v(2\ 2), get \rightarrow \ldots, set \rightarrow \ldots \\ Point_{imp}.set(v(2\ 2)).get &\mapsto\!\!\!\!\rightarrow v(2\ 2) \end{aligned}$$

- 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 [FHM94] and its generalization in the ρCal.

Example 8 (A Fixed Point Object).
Let the symbols rec and f. The fixed-point object Fix_f for f can be easily represented as follows:

$$Fix_f \triangleq rec \rightarrow S \rightarrow f(S.rec)$$

It is not hard to verify that $Fix_f.rec \mapsto\!\!\!\!\rightarrow f(Fix_f.rec)$. This fixed point can be generalized as follows:

$$Fix \triangleq rec \rightarrow S \rightarrow X \rightarrow X(S.rec(X))$$

and its behavior will be as follows:

$$Fix.rec(f) \mapsto\!\!\!\!\rightarrow f(Fix.rec(f))$$

4.1 The Lambda Calculus of Objects

We now present a translation of the Lambda Calculus of Objects of Fisher, Honsell, and Mitchell [FHM94] 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 [GHL98].

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

$$
\begin{array}{rll}
M, N ::= & \lambda(X)M \mid MN \mid X \mid c \mid & \text{Lambda terms} \\
& \langle\rangle \mid & \text{Empty object} \\
& \langle M \leftarrow n = N\rangle \mid & \text{Object override} \\
& \langle M \ \leftrightarrow n = N\rangle \mid & \text{Object extension} \\
& M \Leftarrow n \mid & \text{Method call} \\
& Sel(M, m, N) & \text{Auxiliary term}
\end{array}
$$

Let the operator $\leftarrow\!*$ denote either $\leftarrow$ or $\leftrightarrow$. The small-step semantics is defined by:

$$
\begin{array}{lrcl}
(Beta) & (\lambda(X)M)\,N & \mapsto_{\lambda Obj} & [X/N]M \\
(Sel) & M \Leftarrow m & \mapsto_{\lambda Obj} & Sel(M, m, M) \\
(Succ) & Sel(\langle M \leftarrow\!* \, n = N\rangle, n, P) & \mapsto_{\lambda Obj} & NP \\
(Next) & Sel(\langle M \leftarrow\!* \, n = N\rangle, m, P) & \mapsto_{\lambda Obj} & Sel(M, m, P)
\end{array}
$$

The main operation on objects is method invocation, whose reduction is defined by the (Sel) rule. Sending a message m to an object M containing a method m reduces to $Sel(M, m, M)$. The arguments of Sel in $Sel(M, m, P)$ have the following intuitive meaning (in reverse order):

- P is the receiver (or recipient) of the message;
- m is the message we want to send to the receiver of the message;
- M is (or reduces to) a proper sub-object of the receiver of the message.

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 [FHM94] 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 9 (An object with "self-extension").
Consider the object *Self_ext* [GHL98] defined as follows:

$$Self\_ext \triangleq \langle\langle\rangle \ \leftrightarrow add\_n = \lambda(S)\langle S \ \leftrightarrow n = \lambda(S')1\rangle\rangle$$

If we send the message *add_n* to *Self_ext*, then we get the following computation:

$$
\begin{array}{rcl}
Self\_ext \Leftarrow add\_n & \mapsto_{\lambda Obj} & Sel(Self\_ext, add\_n, Self\_ext) \\
& \mapsto_{\lambda Obj} & (\lambda(S)\langle S \ \leftrightarrow n = \lambda(S')1\rangle)\ Self\_ext \\
& \mapsto_{\lambda Obj} & \langle Self\_ext \ \leftrightarrow n = \lambda(S')1\rangle
\end{array}
$$

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 7. The translation is defined as follows:

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

For instance, Example 10 shows an example of a simple computation in $\lambda\mathcal{O}bj$ and the corresponding translation into ρCal, and Example 11 presents the translation of the *Self_ext* object into ρCal.

Example 10 (A Simple Computation).
Let $Point \triangleq \langle\langle\,\rangle \;\leftrightarrow x = \lambda(S)S \Leftarrow y\rangle \;\leftrightarrow y = \lambda(S)1\rangle$ be a simple diagonal point. Then:

$$\begin{aligned}
Point \Leftarrow x \;&\mapsto_{\lambda\mathcal{O}bj}\; Sel(Point, x, Point) \\
&\mapsto_{\lambda\mathcal{O}bj}\; Sel(\langle\langle\,\rangle \;\leftrightarrow x = \lambda(S)S \Leftarrow y\rangle, x, Point) \\
&\mapsto_{\lambda\mathcal{O}bj}\; (\lambda(S)S \Leftarrow y)Point \\
&\mapsto_{\lambda\mathcal{O}bj}\; Point \Leftarrow y \\
&\mapsto_{\lambda\mathcal{O}bj}\; Sel(Point, y, Point) \\
&\mapsto_{\lambda\mathcal{O}bj}\; (\lambda(S)1)Point \\
&\mapsto_{\lambda\mathcal{O}bj}\; 1
\end{aligned}$$

The above computation in $\lambda\mathcal{O}bj$ can be easily translated as follows into ρCal:
Let $t \triangleq [\![Point]\!] \triangleq x \to S \to S.y, y \to S \to 1$.

$$\begin{aligned}
[\![Point \Leftarrow x]\!] &\triangleq [\![Sel(Point, x, Point)]\!] \\
&\triangleq t.x \;\triangleq\; t\bullet x\bullet t \\
&\equiv (x \to S \to S.y, y \to S \to 1)\bullet x\bullet t \\
&\mapsto\!\!\!\!\to (S \to S.y, null)\bullet t \;=_{\mathbb{T}_{\lambda\mathcal{O}bj}}\; (S \to S.y)\bullet t \\
&\mapsto t.y \;\triangleq\; (x \to S \to S.y, y \to S \to 1)\bullet y\bullet t \\
&\mapsto\!\!\!\!\to (null, S \to 1)\bullet t \;=_{\mathbb{T}_{\lambda\mathcal{O}bj}}\; (S \to 1)\bullet t \\
&\mapsto 1
\end{aligned}$$

Example 11 (Translation of Self_ext).
The object *Self_ext* presented in Example 9 can be translated into the ρCal as follows:

$$t_1 \triangleq [\![Self\_ext]\!] \triangleq add\_n \to S \to (S, n \to S' \to 1)$$

and a simple evaluation of $(t_1.add\_n).n$ is as follows:

$$\begin{array}{rcl}
(t_1.add_n).n & \mapsto & ((S \to (S, n \to S' \to 1))\bullet t_1).n \\
& \mapsto & (t_1, n \to S' \to 1).n \\
& \triangleq & \underbrace{(add_n \to S \to (S, n \to S' \to 1), n \to S' \to 1)}_{t_2}.n \\
& \triangleq & t_2 \bullet n \bullet t_2 \\
& \mapsto & ((add_n \to S \to (S, n \to S' \to 1))\bullet n,\ \ (n \to S' \to 1)\bullet n)\bullet t_2 \\
& \mapsto\!\!\!\!\to & null, (S' \to 1)\bullet t_2\ =_{\mathbb{T}_{\lambda\mathcal{O}bj}}\ (S' \to 1)\bullet t_2 \\
& \mapsto & 1
\end{array}$$

The main result of this section states the relationship between a reduction of a term in $\lambda\mathcal{O}bj$ and the reduction of its ρ-translation in the theory $\mathbb{T}_{\lambda\mathcal{O}bj}$:

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 [AC96] is a calculus where the only existing entities are the objects; it is computationally complete since lambda 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 $a \triangleq [m_i = \varsigma(X)b_i]^{i=1\ldots n}$, then its small-step semantics is:

$$\begin{array}{llll}
(Select) & a.m_j & \mapsto_{\varsigma\mathcal{O}bj}\ [X/a]b_j & j = 1\ldots n \\
(Update)\ a.m_j := \varsigma(X)b & & \mapsto_{\varsigma\mathcal{O}bj}\ [m_i = \varsigma(X)b_i, m_j = \varsigma(X)b]^{i=1\ldots n\setminus\{j\}} & j = 1\ldots n
\end{array}$$

The Translation into ρCal. The translation of an $\varsigma\mathcal{O}bj$-term into a corresponding ρCal-term is quite similar to the one of $\lambda\mathcal{O}bj$, and can be done in the theory $\mathbb{T}_{\varsigma\mathcal{O}bj}$ where the symbol "," is associative and commutative and *null* is its neutral element. Given the function $kill_m$:

$$kill_m : \mathcal{T} \Rightarrow \mathcal{T} \triangleq (X, m \to Y) \to X$$

and the following alias in ρCal:

$$t_1.m := t_2 \triangleq kill_m(t_1), m \to t_2$$

the translation is defined as follows:

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

As a simple example, we present the usual Abadi and Cardelli's encoding of the Point class [AC96].

Example 12 (A Point Class). The object $PClass$ is defined in $\varsigma\mathcal{O}bj$ as follows:

$$\begin{aligned} PClass_{\varsigma\mathcal{O}bj} \triangleq\ & [new = \varsigma(S) \\ & \quad [val = \varsigma(S')(S.preval)(S'), \\ & \quad\ get = \varsigma(S')(S.preget)(S'), \\ & \quad\ set = \varsigma(S')(S.preset)(S')] \\ & preval = \varsigma(S)\lambda(S')v(1\ 1), \\ & preget = \varsigma(S)\lambda(S')S'.val, \\ & preset = \varsigma(S)\lambda(S')\lambda(N)S.val := \varsigma(S'')N] \end{aligned}$$

and it is translated into the ρCal as follows:

$$\begin{aligned} PClass_{\rho Cal} \triangleq\ & new \to S \to \\ & \quad (val \to S' \to (S.preval)\bullet S', \\ & \quad\ get \to S' \to (S.preget)\bullet S', \\ & \quad\ set \to S' \to (S.preset)\bullet S'), \\ & preval \to S \to S' \to v(1\ 1), \\ & preget \to S \to S' \to S'.val, \\ & preset \to S \to S' \to v(X\ Y) \to (S'.val := S'' \to v(X\ Y)) \end{aligned}$$

It is not hard to verify that if we send the message new to $PClass_{\rho Cal}$ we obtain a point, *i.e.*:

$$\begin{aligned} PClass_{\rho Cal}.new \twoheadrightarrow\ & Point_{\rho Cal} \\ \triangleq\ & val \to S \to v(1\ 1), \\ & get \to S \to S.val, \\ & set \to S \to v(X\ Y) \to (S.val := S' \to v(X\ Y)) \end{aligned}$$

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 $\varsigma\mathcal{O}bj$:

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

and the alias in $\varsigma\mathcal{O}bj$

$$p \circ q \triangleq (p.arg := \varsigma(S)q).val$$

which represents the encoding of the function application.

Example 13 (Another Fixed-Point Object). Let the symbols arg, val and f. The generic fixed-point object Fix in $\varsigma\mathcal{O}bj$:

$$\begin{aligned} Fix \triangleq\ & [arg = \varsigma(S)S.arg, \\ & \quad val = \varsigma(S)((S.arg).arg := \varsigma(S')S.arg).val] \end{aligned}$$

can be translated into ρCal as follows:

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

Using the following aliases in ρCal:

$$\begin{array}{rcll} Fix_f & \triangleq & Fix.arg := S' \to f & \text{for a fresh } S' \\ t_1 \circ t_2 & \triangleq & (t_1.arg := S' \to t_2).val & \text{for a fresh } S' \end{array}$$

we can prove that:

$$\begin{array}{rcl} Fix \circ f & \equiv & Fix_f.val \\ & \mapsto\!\!\!\!\to & ((Fix_f.arg).arg := S' \to Fix_f.val).val \\ & \mapsto\!\!\!\!\to & (f.arg := S' \to Fix \circ f).val \\ & \equiv & f \circ (Fix \circ f) \end{array}$$

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

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

The following examples show that the expressive power of ρCal is strictly stronger than the two previous calculi of objects as they cannot be translated neither in $\lambda\mathcal{O}bj$ nor in $\varsigma\mathcal{O}bj$. Thanks to the higher-order nature of the syntax of the Rho Calculus we can easily consider "labels" and "methods" as *first-class entities* that can be passed as function arguments.

Example 14 (The Daemon and the Para object).

1. Let the symbol set. The object $Daemon$ is defined as follows:

$$Daemon \triangleq set \to S \to X \to (X, set \to S' \to Y \to (Y, S'))$$

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. A simple application of the method gives:

$$\begin{array}{rcl} Daemon.set(x \to S \to 3) & \triangleq & Daemon \bullet set \bullet Daemon \bullet (x \to S \to 3) \\ & \mapsto & (S \to X \to (X, set \to S' \to Y \to (Y, S'))) \bullet Daemon \bullet (x \to S \to 3) \\ & \mapsto & (X \to (X, set \to S' \to Y \to (Y, S'))) \bullet (x \to S \to 3) \\ & \mapsto & \underbrace{x \to S \to 3, set \to S' \to Y \to (Y, S')}_{obj} \end{array}$$

and

$$obj.set(y \to S \to 4) \mapsto\!\!\!\!\to y \to S \to 4, x \to S \to 3, set \to S' \to Y \to (Y, S')$$

2. Let a symbol par. The object $Para$ is defined as follows:

$$Para \triangleq a \to S \to b, par(X) \to S \to S.X$$

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:

$$\begin{array}{rcl} Para.(par(a)) & \triangleq & Para \bullet (par(a)) \bullet Para \\ & \mapsto & ((a \to S \to b) \bullet (par(a)), (par(X) \to S \to S.X) \bullet (par(a))) \bullet Para \\ & \mapsto\!\!\!\!\to & (S \to S.a) \bullet Para \\ & \mapsto & Para.a \\ & \mapsto\!\!\!\!\to & b \end{array}$$

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 [CK99a,CK99b,Cir00], 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 ρCal based language ELAN [DK00]. More generally, rewrite based languages like ASF+SDF, CafeOBJ, Maude or ELAN, could benefit from a ρCal 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 applications of the framework are numerous; a track that we have not yet mention in this paper concerns encoding concurrency in the spirit of the early work of Viry [Vir96].

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

Acknowledgement.

We thank Roberto Bruni and David Wolfram for their precise and useful comments on the first version of the rewriting calculus. We would like also to thank all the members of the ELAN group for their comments and interactions on the topics of the Rho Calculus.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[AKPS94] 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.

[Bar84] H. Barendregt. *Lambda Calculus: its Syntax and Semantics*. North Holland, 1984.

[BKK+98] 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.

[BT88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proc. of LICS*, pages 82–90, 1988.

[Bür89] H.-J. Bürckert. Matching — A special case of unification? *Journal of Symbolic Computation*, 8(5):523–536, 1989.

[CELM96] 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.

[Cir00] Horatiu Cirstea. *Calcul de réécriture : fondements et applications*. PhD thesis, Université Henri Poincaré - Nancy I, 2000. to appear.

[CK99a] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In *Frontiers of Combining Systems 2*, pages 95–120. Wiley, 1999.

[CK99b] H. Cirstea and C. Kirchner. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, 1999.

[DHK96] A. Deursen, J. Heering, and P. Klint. *Language Prototyping*. World Scientific, 1996. ISBN 981-02-2732-9.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[DHKP96] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *Proc. of JICSLP*. The MIT press, 1996.

[DK00] Hubert Dubois and Hélène Kirchner. Objects, rules and strategies in ELAN. Submitted, 2000.

[Dow94] G. Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.

[FH86] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(1):189–200, 1986.

[FHM94] 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.

[FN97] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.

[GBT89] J. Gallier and V. Breazu-Tannen. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *Proc. of ICALP*, volume 372 of *LNCS*, pages 137–150. Springer-Verlag, 1989.

[GHL98] 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.

[GKK+87] 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.

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

[HO82] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

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

[Hul79] J.-M. Hullot. Associative-commutative pattern matching. In *Proc. of IJCAI*, 1979.

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

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

[KK99] Claude Kirchner and Hélène Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKV95] 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.

[KMP77] D. E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

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

[Lav87] A. Laville. Lazy pattern matching in the ML language. In *Proc. FCT & TCS*, volume 287 of *LNCS*, pages 400–419. Springer-Verlag, 1987.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984.

[Mil91] 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. Also in *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mil96] D. Miller. Forum: A multiple-conclusion meta-logic. *Theoretical Computer Science*, 110(1):201–232, 1996.

[Nip89] T. Nipkow. Combining matching algorithms: The regular case. In *Proc. of RTA*, volume 355 of *LNCS*, pages 343–358. Springer-Verlag, 1989.

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

[Oka89] 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.

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

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

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[Rin96] Ch. Ringeissen. Combining Decision Algorithms for Matching in the Union of Disjoint Equational Theories. *Information and Computation*, 126(2):144–160, 1996.

[Tak89] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 7:113–123, 1989.

[Vir96] P. Viry. Input/Output for ELAN. In *Proc. of WRLA*, volume 4. Electronic Notes in Theoretical Computer Science, 1996.

[vO90] V. van Oostrom. Lambda calculus with patterns. Technical Report IR-228, Vrije Universiteit, 1990.

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

A For Referees

A.1 Substitutions

As for any calculus involving binders like the lambda calculus, α-conversion should be used in order to obtain a correct substitution calculus and the first-order substitution (also called grafting) is not directly suitable for the Rho Calculus. We consider the usual notions of α-conversion and higher-order substitution as defined for example in [DHK00] and briefly presented in what follows.

Definition 5 (Free Variables). *The set of Free Variables is inductively defined as follows:*

$$\begin{aligned} FV(a) &\triangleq \emptyset \\ FV(X) &\triangleq X \\ FV(t_1 \to t_2) &\triangleq FV(t_2) \setminus FV(t_1) \\ FV(t_1 \bullet t_2) &\triangleq FV(t_1) \cup FV(t_2) \\ FV(t_1, t_2) &\triangleq FV(t_1) \cup FV(t_2) \end{aligned}$$

Definition 6 (α-conversion). *Given a set $\mathcal{X} \subseteq \mathcal{V}$ of variables, the application $\alpha_{\mathcal{X}}$ (called α-conversion) is defined by:*

$$\begin{aligned} \alpha_{\mathcal{X}}(a) &\triangleq a \\ \alpha_{\mathcal{X}}(X) &\triangleq X \\ \alpha_{\mathcal{X}}(t_1 \bullet t_2) &\triangleq \alpha_{\mathcal{X}}(t_1) \bullet \alpha_{\mathcal{X}}(t_2) \\ \alpha_{\mathcal{X}}(t_1 \to t_2) &\triangleq \alpha_{\mathcal{X}}(t_1) \to \alpha_{\mathcal{X}}(t_2) \\ &\quad if\ FV(t_1) \cap \mathcal{X} = \emptyset, \\ \alpha_{\mathcal{X}}(t_1 \to t_2) &\triangleq [X_i \mapsto Y_i]\alpha_{\mathcal{X}}(t_1) \to [X_i \mapsto Y_i]\alpha_{\mathcal{X}}(t_2) \\ &\quad if\ X_i \in FV(t_1) \cap \mathcal{X}, \end{aligned}$$

where Y_i are "fresh" variables and $[X \mapsto Y]$ denotes the replacement (grafting) of the variable X by the variable Y in the term on which it is applied.

This allows us to define the usual substitution operation:

Definition 7 (Substitution). *The application of a substitution $\sigma \equiv [X_1/t_1 \dots X_n/t_n]$ is structurally defined by:*

$$\begin{aligned} \sigma a &\triangleq a \\ \sigma X_i &\triangleq t_i \\ \sigma(u \bullet v) &\triangleq (\sigma u) \bullet (\sigma v) \\ \sigma(u, v) &\triangleq (\sigma u), (\sigma v) \\ \sigma(u \to v) &\triangleq (\sigma u') \to (\sigma v') \end{aligned}$$

where we consider that $Y_i \in FV(u)$, Z_i are fresh variables, i.e. $\sigma Z_i = Z_i$, Z_i do not occur in u and v, that for any $Y \in FV(u) \cup FV(v)$, $Z_i \notin FV(\sigma Y)$, and that u', v' are defined by:

$$\begin{aligned} u' &\triangleq [Y_i \mapsto Z_i]\alpha_{FV(u) \cup Var(\sigma)}(u) \\ v' &\triangleq [Y_i \mapsto Z_i]\alpha_{FV(u) \cup Var(\sigma)}(v) \end{aligned}$$

Recall that $[X_1/t_1 \dots X_n/t_n]$ denotes the simultaneous substitution of the variables $X_1 \dots X_n$ by the terms $t_1 \dots t_n$ and not the composition $[X_1/t_1] \dots [X_n/t_n]$.

We consider a total order $\prec$ on the set of substitutions. Such orderings always exist: one can for example take the lexicographic ordering on the flattened representation of the substitutions. The ordering on substitutions can be seen as a parameter of the ρCal and thus can be customized by the user.

Example 15 (Lexicographic ordering on σ's). We list some examples of substitutions ordered using the ordering suggested above:

1. $[X/a\ Y/b] \prec [X/b\ Y/a]$ since $XaYb \prec_{lex} XbYa$
2. $[X/a\ Y/b] \prec [X/a\ Y/b\ Z/c]$ since $XaYb \prec_{lex} XaYbZc$

A.2 Properties

Theorem 4 (Confluence in $\mathbb{T}_\emptyset$). *Given a term t_1 such that all its rewrite rules contain no abstraction in the left-hand side, if $t_1 \mapsto\!\!\!\!\twoheadrightarrow t_2$ and $t_1 \mapsto\!\!\!\!\twoheadrightarrow t_3$ then there exists a term t_4 such that $t_2 \mapsto\!\!\!\!\twoheadrightarrow t_4$ and $t_3 \mapsto\!\!\!\!\twoheadrightarrow t_4$.*

The proof, close to the one given for the previous version of the calculus [Cir00], is done by using an approach similar to the "parallel reduction" due to *Tait & Martin-Löf* [Mar84,Tak89]. When a more elaborated theory $\mathbb{T}$ is used we have to check the coherence between the theory and the evaluation rules of the $\rho Cal_{\mathbb{T}}$.

Conjecture 1. The $\mathbb{T}_{\lambda\mathcal{O}bj}$ and the $\mathbb{T}_{\varsigma\mathcal{O}bj}$ are confluent.

Theorem 5 (Translation of $\lambda\mathcal{O}bj$ into ρCal). *If $M \mapsto_{\lambda\mathcal{O}bj} N$, then $[\![M]\!] \mapsto\!\!\!\!\twoheadrightarrow_{\mathbb{T}_{\lambda\mathcal{O}bj}} [\![N]\!]$.*

Proof. We consider all the rules describing the operational semantics of $\lambda\mathcal{O}bj$ and we show that a corresponding reduction is obtained for the translation of the terms from the two sides of each rule:

- $(Beta)\quad (\lambda X.M)\,N \mapsto_{\lambda\mathcal{O}bj} [X/N]M$. Then we have:
  $[\![(\lambda X.M)\,N]\!] \triangleq (X \to [\![M]\!])\bullet N \overset{\rho}{\mapsto} [X/[\![N]\!]][\![M]\!] \triangleq [\![[X/N]M]\!]$
- $(Sel)\quad M \Leftarrow m \mapsto_{\lambda\mathcal{O}bj} Sel(M,m,M)$. Then we have:
  $[\![M \Leftarrow m]\!] \triangleq [\![M]\!].[\![m]\!] \triangleq [\![Sel(M,m,M)]\!]$
- $(Succ)\quad Sel(\langle M \leftarrow n = N\rangle, n, P) \mapsto_{\lambda\mathcal{O}bj} NP$. Then we have:
  $[\![Sel(\langle M \leftarrow n = N\rangle, n, P)]\!] \triangleq [\![\langle M \leftarrow n = N\rangle]\!]\bullet[\![n]\!]\bullet[\![P]\!]$.
  It is easy to check that $kill_{[\![n]\!]}([\![M]\!])\bullet[\![n]\!] \mapsto\!\!\!\!\twoheadrightarrow null$ and we have the following two cases:
  1. *n is already a method of M*
     $[\![\langle M \leftarrow n = N\rangle]\!]\bullet[\![n]\!]\bullet[\![P]\!] \triangleq (kill_n([\![M]\!]), [\![n]\!] \to [\![N]\!])\bullet[\![n]\!]\bullet[\![P]\!] \overset{\epsilon}{\mapsto}$
     $(kill_n([\![M]\!])\bullet[\![n]\!], ([\![n]\!] \to [\![N]\!])\bullet[\![n]\!])\bullet[\![P]\!] \mapsto\!\!\!\!\twoheadrightarrow (null, [\![N]\!])\bullet[\![P]\!] =_{\mathbb{T}_{\lambda\mathcal{O}bj}} [\![N]\!]\bullet[\![P]\!] \triangleq [\![NP]\!]$
  2. *n is not a method of M*
     $[\![\langle M \leftarrow n = N\rangle]\!]\bullet[\![n]\!]\bullet[\![P]\!] \triangleq ([\![M]\!], [\![n]\!] \to [\![N]\!])\bullet[\![n]\!]\bullet[\![P]\!] \overset{\epsilon}{\mapsto} ([\![M]\!]\bullet[\![n]\!], ([\![n]\!] \to [\![N]\!])\bullet[\![n]\!])\bullet[\![P]\!] \overset{\rho}{\mapsto}$
     $(null, [\![N]\!])\bullet[\![P]\!] =_{\mathbb{T}_{\lambda\mathcal{O}bj}} [\![N]\!]\bullet[\![P]\!] \triangleq [\![NP]\!]$
- $(Next)\quad Sel(\langle M \leftarrow n = N\rangle, m, P) \mapsto_{\lambda\mathcal{O}bj} Sel(M,m,P)$. Then we have:
  $[\![Sel(\langle M \leftarrow n = N\rangle, m, P)]\!] \triangleq [\![\langle M \leftarrow n = N\rangle]\!]\bullet[\![m]\!]\bullet[\![P]\!] \triangleq ([\![M]\!], [\![n]\!] \to [\![N]\!])\bullet[\![m]\!]\bullet[\![P]\!] \overset{\epsilon}{\mapsto}$
  $([\![M]\!]\bullet[\![m]\!], ([\![n]\!] \to [\![N]\!])\bullet[\![m]\!])\bullet[\![P]\!] \overset{\rho}{\mapsto} ([\![M]\!]\bullet[\![m]\!], null)\bullet[\![P]\!] =_{\mathbb{T}_{\lambda\mathcal{O}bj}} ([\![M]\!]\bullet[\![m]\!])\bullet[\![P]\!] \triangleq [\![Sel(M,m,P)]\!]$

□

Theorem 6 (Translation of $\varsigma\mathcal{O}bj$ into ρCal). *If $M \mapsto_{\varsigma\mathcal{O}bj} N$, then $[\![M]\!] \mapsto\!\!\!\!\twoheadrightarrow_{\mathbb{T}_{\varsigma\mathcal{O}bj}} [\![N]\!]$.*

Proof. We consider all the rules describing the operational semantics of $\varsigma\mathcal{O}bj$ and we show that a corresponding reduction is obtained for the translation of the terms from the two sides of each rule:
Let $a \triangleq [m_i = \varsigma(X)b_i]^{i=1\ldots n}$.

- $(Send)\quad a.m_j \mapsto_{\varsigma\mathcal{O}bj} b_j[X/a] \quad j = 1\ldots n$. Then we have:
  $[\![a.m_j]\!] \triangleq [\![[m_i = \varsigma(X)b_i]^{i=1\ldots n}.m_j]\!] \triangleq ([\![m_i]\!] \to [\![X]\!] \to [\![b_i]\!])^{i=1\ldots n}.[\![m_j]\!] \triangleq ([\![m_i]\!] \to [\![X]\!] \to [\![b_i]\!])^{i=1\ldots n}\bullet[\![m_j]\!]\bullet[\![a]\!]$
  $\overset{\rho}{\mapsto\!\!\!\!\twoheadrightarrow} (null, \ldots [\![X]\!] \to [\![b_j]\!], \ldots null)\bullet[\![a]\!] =_{\mathbb{T}_{\lambda\mathcal{O}bj}} ([\![X]\!] \to [\![b_j]\!])\bullet[\![a]\!] \overset{\rho}{\mapsto} [\![b_j]\!][[\![X]\!]/[\![a]\!]] \triangleq [\![b_j[X/a]]\!]$
- $(Update)\quad a.m_j = \varsigma(X)b \mapsto_{\varsigma\mathcal{O}bj} [m_i = \varsigma(X)b_i, m_j = \varsigma(X)b]^{i=1\ldots n\setminus j}\ j = 1\ldots n$. Then we have:
  $kill_{[\![m_j]\!]}([\![a]\!]) \triangleq kill_{[\![m_j]\!]}([\![m_i]\!] \to [\![X]\!] \to [\![b_i]\!])^{i=1\ldots n} \mapsto\!\!\!\!\twoheadrightarrow ([\![m_i]\!] \to [\![X]\!] \to [\![b_i]\!])^{i=1\ldots n\setminus j}$
  and
  $[\![a.m_j = \varsigma(X)b]\!] \triangleq (kill_{m_j}([\![a]\!]), [\![m_j]\!] \to [\![X]\!] \to [\![b]\!]) \mapsto\!\!\!\!\twoheadrightarrow ([\![m_i]\!] \to [\![X]\!] \to [\![b_i]\!])^{i=1\ldots n\setminus j}, [\![m_j]\!] \to [\![X]\!] \to [\![b]\!]) \triangleq$
  $[\![[m_i = \varsigma(X)b_i, m_j = \varsigma(X)b]^{i=1\ldots n\setminus j}]\!]$

□