LISP AND SYMBOLIC COMPUTATION, 9, 153?? (1996)


# A Semantics of Introspection in a Reflective Prototype-Based Language

JACQUES MALENFANT malenfan@iro.umontreal.ca
*Département d'informatique et recherche opérationnelle, Université de Montréal, C.P. 6128, Succursale A, Montréal, Québec, CANADA H3C 3J7*

CHRISTOPHE DONY dony@lirmm.fr
*LIRMM, Université de Montpellier, 161 rue Ada, 34392 Montpellier Cedex 5, France*

PIERRE COINTE cointe@emn.fr
*École des Mines de Nantes, 4 rue Alfred Kastler, La Chantrerie, 44070 Nantes Cedex 03, France*

**Abstract.** In Malenfant et al. [19], we have described a reflective model for a prototype-based language based on the *lookup* ∘ *apply* reflective introspection protocol. In this paper, we augment our previous protocol by converting it to handle continuations reified as first-class objects. First-class continuations provide much more control over the current computation; during the introspection phase fired by message sending, they make it possible not only to change the behavior of the program for that message but also for the entire future computation. In this paper, we provide this introspection protocol with a formal semantics. This result is obtained by exhibiting a mapping $\Delta$ from program configurations to priority rewrite systems (PRS) as well as a mapping from message expressions to ground first-order terms used to query the PRS. Other advantages of this approach are: to ensure the termination of the introspection using the smallest set of formally justified conditions and to provide a clear declarative account of this reflective protocol. The PRS also appears as a meta-level to the base language, independent of the implementation, but from which we derive fundamental clues to obtain an efficient language processor. By our new model, we finally highlight the link between reflection in object-oriented languages and the one originally proposed by 3-Lisp [24], although object-orientation provides reusability to reflection, making it easier to use.

**Keywords:** Procedural reflection, object-oriented programming, reflective towers, priority rewrite systems, semantics.

## 1. Introduction

Reflection, understood as the construction of self-aware systems, is a persistent source of challenge. The tremendous potential for new applications ensure a continuous quest for understanding its foundations. The goal of this paper is to propose a minimal model of behavioral reflection for a prototype-based language, to study it in depth and to provide it with a formal semantics using the theory of priority rewrite systems (PRS) [2], [21], [5]. We propose a complete prototype-based language, but our formal study is restricted to the reflective protocol only; hence our

results can be applied to other reflective object-oriented programming languages, provided they use a similar reflective protocol.

Object-oriented programming is dominated by the class and metaclass approach which provides it with a highly satisfactory solution to the problem of *structural reflection*, the complete reification of data structures and programs as first-class entities. The problem of *behavioral reflection*, dealing with the reification of objects' execution, is not yet established on similar firm grounds. The main approach currently investigated to represent the behavioral properties of objects is based on meta-objects [17], [27], [9] but several problems are still open:

1. How can we overcome the potential infinite regression when using meta-objects?
2. What protocol should be used to connect behavioral meta-objects and the evaluator's self-representation?
3. How should we represent the evaluator's data structures and execution?
4. What is the relationship between structural and behavioral reflection? Should the behavioral meta-object of an object be the same as its structural one (e.g., its class)?

We implement reflection in a prototype-based language in order to avoid the unnecessary complexity of classes and postpone the last question until we fully understand behavioral reflection. We choose to implement meta-object based behavioral reflection and to reify message passing using the *lookup* ∘ *apply* reflective protocol. Hence, we use and extend existing ideas in order to push them to their limit. To concentrate on the central issues, our proposal is a minimalist one: we work on a minimal prototype-based language proposed in our previous work [8], [19], to which structural reflection is provided in a minimal way, just to make behavioral reflection work properly.

The goal of behavioral reflection is to give to the user complete control over the current computation at run-time. To achieve this goal, we augment the reflective protocol to handle continuations reified as first-class objects. Our new *lookup* ∘ *apply* reflective protocol augmented with a reification of continuations provides a reflective programming model that matches those of 3-Lisp and reflective extensions of Scheme [12], [13]. *Lookup* and *apply* methods are now able to examine, modify and otherwise deal with continuations at run-time. The outcome is that we can adapt the language to particular programs, we can adapt programs' behavior to their current execution state and finally, we can perform all sorts of self-optimizations dynamically.

The outline of the paper is the following. In the next section, we present our reflective prototype-based language, including sufficient structural reflection capabilities to enable the implementation of behavioral reflection. In Section 3, we give the formal semantics for the reflective protocol. In Section 4, we discuss the object's general behavior, the implementation of the resulting language, and the status of rewriting according to the implementation. Section 5 compares our approach with related work. We then conclude and discuss future work.

## 2. Reflective Prototypes

### 2.1. A Minimal Language

In [8], we have proposed that a prototype-based language should be implemented on the basis of the following principles:

**P1**: A prototype is represented as a collection of slots. A slot can represent either a data value (data slot) or a method (method slot).

**P2**: Message passing is the only means to activate a prototype and slot names are used as selectors in messages. No difference is made between data slots and method slots, both are accessed through message passing [25].

**P3**: A prototype is constructed as an extension of an existing prototype using parent-of implicit delegation links; the prototype called `ROOT` is used as root of parent-of delegation hierarchies.

**P4**: The structure of a prototype is immutable, i.e. one cannot add or retract a slot within an object; this allow encapsulation of objects to be implemented effectively by preventing malicious users from dynamically adding public accessors to private information.

**P5**: *newInitials(p,initform)* is the first primitive function to create new objects with a fixed set of slots with initial values; this primitive is invoked by a message `(p 'new-initials initform)`[1] where the receiver `p` is the parent of the new object.

**P6**: *clone(p)* is an alternative primitive function to create new prototypes by copying existing ones; this primitive is invoked by a message `(p clone)` where `p` is the prototype to be copied.

For reasons out of the scope of this paper, the object `ROOT` is defined as a root of implicit delegation hierarchies and gets as methods all the primitive functions of the language (*newInitials* and *clone*, see [8] for more details). Finally, we assume that prototypes have only one parent; this restriction could be relaxed, but multiple parents add nothing to our study except an unnecessary complexity.

### 2.2. Structural Reflection

In class-based languages, classes and metaclasses are first-class objects implementing structural reflection but in prototype-based languages, there are no more classes to deal with the structure of objects; an alternative must be sought to obtain similar capabilities. In fact, prototypes are not easily amenable to structural reflection [18]. To link a prototype to another one that describes its structure goes against the principles of prototype-based programming in the most fundamental way. However, prototypes still provide a simple object-oriented model that allows us to study behavioral reflection in depth.

Because studying behavioral reflection needs only little structural reflection capabilities, we use very limited ones: access to the structural information about individual objects and a reification of methods as objects. We identified [18], [11] five primitive access functions: *size(p)* to get the size of a prototype *p*, *nameAt(p,i)* to get the name of its *ith* slot, *contentsAt(p,i)* to get the content of its *ith* slot, *contentsAtPut(p,i,v)* to set the content of its *ith* slot, and *isMethodAt(p,i)* to test whether its *ith* slot is a method slot or a data slot. These primitive functions are represented as methods in the language, themselves reified as the objects: `SizeP`, `NameAtP`, `ContentsAtP`, `ContentsAtPutP`, and `IsMethodAtP`. The object `ROOT` gets these methods, to which it points through its method slots: `size`, `name-at`, `contents-at`, `contents-at-put`, and `is-method-at` respectively.

Note that we do not consider this proposal as a definitive solution to structural reflection. We simply use it as a working one in order to proceed with behavioral reflection. For simplicity, we make objects themselves responsible for responding to the "reflective" messages.

### 2.3. Behavioral Reflection

A behavioral reflection model must describe the behavior of objects using other objects and it must provide a method invocation protocol that allows the user to intervene in the current execution in order to modify the course of events, i.e. to reflect. To describe the behavior of objects, we associate a meta-object to each of them. This meta-object defines how its associated object reacts when it receives a message. Since the object-oriented model of computation is based on message passing, it is usual to make message sending the vantage point where programs can shift into a reflecting phase. The standard way to achieve that is by making visible the two main operations done by the evaluator when a message is sent: the lookup and the application of the method. This is the traditional equation where message sending is viewed as the composition (from left to right) of a lookup and an apply:

$$\textit{message execution} = \textit{lookup} \circ \textit{apply}$$

These operations can be implemented by methods reified as objects in the language, allowing the user to redefine them in order to perform reflective computations. Each message passing operation `(o 'selector a1 a2 a3)` is replaced by a reflective introspection in three phases: (1) find the meta-object of the receiver `o`, (2) send it a lookup message for the selector `'selector` in the receiving object, which yields a method object (3) to which is sent an apply message to execute in in the context of the receiving object with the message arguments. Hence, coarsely speaking, the reflective introspection rule is:

```
(o 'selector a1 a2 a3) ⇒ (((o* metaObject)
                           'lookup 'selector o*)
                           'apply-to o* '(a1* a2* a3*) k)
```

where `k` is the continuation, captured by the evaluator, representing the rest of the computation at the time the reflective introspection is made. When applying the reflective introspection rule, the receiver `o`, and the arguments `a1`, `a2` and `a3` of the message may be evaluated. We denote this evaluation by a trailing `*`. Because we do not address this issue, we only assume that the respective results of these evaluations are first-class objects in order to make the reflective introspection rule work properly. Otherwise, we put no constraints on the way the evaluator treats the receiver and the selector or on the mode it uses to evaluate arguments. For the sake of brevity, we will drop the trailing * in the rest of the paper.

Note that each operation in the reflective protocol can lead to potential infinite meta-regression. Since all three are represented as message sending, the same introspection rule can be applied to each of them *ad infinitum*. Section 3 is devoted to the way the language prevents this from happening.

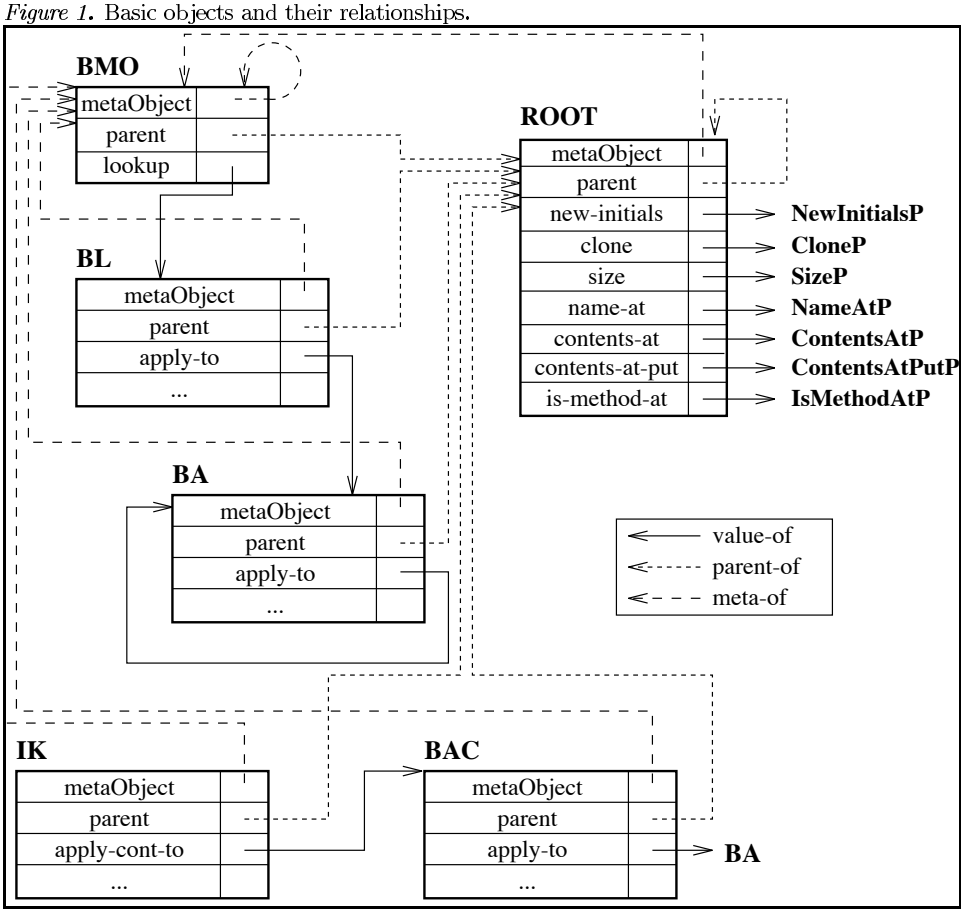
*Data slots*

Because data slots and method slots are both accessed using messages, the protocol must cope with data as well as methods. In both cases, the lookup phase must return an object able to respond to an *apply* message. Because data slots contain only values, we must bridge the gap between values and the object expected by the reflective protocol. Representing data as objects able to respond to *apply* messages is an appealing solution, but it leads immediately to an infinite meta-regression when trying to represent data. We have decided to force the lookup phase to return an object when a message accesses a data slot. This object is created on the fly, in a lazy fashion; it responds to *apply* messages by simply returning the value of its corresponding data slot.

*First-class continuations*

First-class continuations now represent a long tradition, especially in the functional programming community (Lisp, Scheme, ML, ...). Continuations represent the (default) future of the computation at a given point in the execution of the program. Following the Scheme tradition [12], continuations could be represented as method objects (closures) in our language while in the ML one, we could use first order data structures [1]. However, we prefer not to deal explicitly with such issues here and to give as much freedom as possible to implementors. For this reason, we make almost no assumption about the way continuations are represented and about the way they are created. We simply assume that they are objects, invoked by sending them an apply message, such as in `(k 'apply-cont-to o)` where `k` is a continuation object and `o` is the object resulting from the previous expression.

*Figure 1.* Basic objects and their relationships.




*Kernel Prototypes*

The impact of the method invocation protocol is summarized by the following principles added to the prototype-based language:

**P7**: Every object has a meta-object that is able to respond to lookup messages; meta-objects can be shared among several objects.

**P8**: Meta-objects respond to lookup messages by returning method objects that are able to respond to apply messages.

Meta-objects and the above method invocation protocol raise four fundamental problems: (1) an infinite regression of meta-objects may arise along the *meta-of* link between an object and its meta-object, while (2) a basic lookup method, (3) a basic apply method as well as (4) a basic method for applying continuations must be

provided. Our model solves the problem (1) of the potential infinite regression by introducing a basic meta-object, called `BasicMetaObject`, which is its own meta-object, and which defines the standard behavior for objects in the system. This circularity of the *meta-of* link closes the meta-regression on `BasicMetaObject`, in a similar way as the *instance-of* link is closed over `Class` in ObjVLisp [3].

Because of the method invocation protocol, `BasicMetaObject`, as any other meta-object in the system, must be able to respond to lookup message. In fact, since it defines the standard behavior, its lookup method is the primitive lookup function reified as a method in the language (2), which we call `BasicLookup`. Reifying the primitive lookup function as `BasicLookup` requires the introduction of its apply method to respect the method invocation protocol. We assume that its apply method is the primitive apply function also reified as a method in the language (3), which we call `BasicApply` and construct it to have itself as its own apply method. In the same way, reifying continuations require the introduction of a primitive function to apply them, reified in the language (4), which we call `BasicApplyCont`, and construct it to have `BasicApply` as apply method. These solutions add four principles to the reflective prototype-based language:

**P9**: `BasicMetaObject` is the kernel meta-object of the system; it considers itself as its own meta-object.

**P10**: `BasicLookup` is the kernel lookup method of the system; it represents the primitive lookup function (*bl*) reified as a method object in the language.

**P11**: `BasicApply` is the kernel apply method of the system; it represents the primitive apply function (*baf*) reified as a method object in the language.

**P12**: `BasicApplyCont` is the kernel method of the system to apply continuations; it represents the primitive function to apply continuations (*bac*), reified as a method object in the language.

The kernel of our model is constructed around six objects: `BasicMetaObject`, `BasicLookup`, `BasicApply`, `BasicApplyCont`, `IK` and `ROOT` (see §2). `IK` is an object playing the role of the identity continuation; it also holds the basic methods for continuations, such as `BasicApplyCont`, and can serve as the root of the continuation objects hierarchies. The Figure 1 illustrates the kernel objects as well as their relationships.

## 3. A formal semantics for reflective introspection

In this section, we recast our model under the theory of priority rewrite systems (PRS) in order to give a formal semantics to the *lookup* ∘ *apply* reflective introspection protocol. We first recall the basics of PRS and then show how to map a program configuration $\mathcal{C}$ (set of objects) to a PRS $\Delta(\mathcal{C})$ shown to be sound and complete.

### 3.1. Priority Rewrite Systems

Rewrite systems [5] are means to compute by reducing expressions using a set of rewrite rules. The computation performs rewriting steps beginning with a starting expression and ending with a reduct, an expression to which no more rewriting step can be applied. Rewrite systems represent expressions with first-order terms, which are formally defined as follows:

**Definition 1 (Syntax)** *Given the set* $\mathcal{F} = \bigcup_{n\geq 0}\mathcal{F}_n$ *of function symbols — called a (finitary) vocabulary or signature — and a (denumerable) set* $\mathcal{H}$ *of variable symbols, the set of* **(first-order) terms** $\mathcal{T}(\mathcal{F},\mathcal{H})$ *is the smallest set containing* $\mathcal{H}$ *such that* $f(t_1,t_2,\ldots,t_n) \in \mathcal{T}(\mathcal{F},\mathcal{H})$ *whenever* $f \in \mathcal{F}_n$ *and* $t_i \in \mathcal{T}(\mathcal{F},\mathcal{H})$ *for* $i = 1,\ldots,n$.

Given $t = f(t_1,\ldots,t_n)$, $t_1,\ldots,t_n$ are called subterms of $t$, and so are their respective subterms. A term $t$ is viewed as a tree where any of its subterm $s$ is located by a sequence $p$ of integers specifying the path from the root to the subterm $s$; we call $p$ a *position* in the term $t$. The term obtained by replacing the subterm at position $p$ in term $t$ by $s$ is denoted $t[s]_p$. A *context* is a term $u$ with a "hole" at position $p$ where the replacement takes place.

**Definition 2** *A* **substitution** *is a replacement operation that transforms a term* $t$ *to a term* $s$ *by mapping variables in* $t$ *to terms. This mapping is written out as* $\{x_1 \mapsto t_1,\ldots,x_m \mapsto t_m\}$ *when there are only finitely many variables not mapped to themselves. More formally, a substitution is a function denoted* $\sigma$ *from* $\mathcal{H}$ *to* $\mathcal{T}(\mathcal{F},\mathcal{H})$ *extended to a function from* $\mathcal{T}$ *to itself in such a way that* $f(t_1,\ldots,t_n)\sigma = f(t_1\sigma,\ldots,t_n\sigma)$ *for each* $f \in \mathcal{F}$ *and for all terms* $t_i \in \mathcal{T}$.

A term t *matches* (or *unifies* with) a term s if $s\sigma = t$ for some substitution $\sigma$, which is then also called a *unifier* of $s$ and $t$. In general, there exist many unifiers of two terms. For example, the terms $a(x_1)$ and $a(x_2)$ will match for all unifiers that map $x_1$ and $x_2$ to the same term. The *most general unifier* of terms $s$ and $t$ is the substitution $\sigma$ such that, for all unifiers $\nu$ of $s$ and $t$, there exists a substitution $\sigma'$ such that $\nu = \sigma \circ \sigma'$, the composition of $\sigma$ and $\sigma'$.

**Definition 3** *A* **rewrite relation** $\rightarrow$ *is a binary relation over a set of terms* $\mathcal{T}$ *closed under context application and under substitution. A rewrite relation is specified by a set* $\mathcal{R} \subseteq (\mathcal{T}\times\mathcal{T})$ *of rewrite rules of the form* $t \rightarrow s$, *each of which specifies how a term* $t$ *can be rewritten to a term* $s$. *Closeness under context application and substitution means that* $t \rightarrow s$ *implies* $u[t\sigma]_p \rightarrow u[s\sigma]_p$, *for all terms* $s,t \in \mathcal{T}$, *contexts* $u$, *positions* $p$ *and substitutions* $\sigma$.

Each reduction step involves the application of a rewrite rule on the current term $t$ that is done in two steps 1) find a redex $r$ at position $p$ in $t$ and a rewrite rule $u \rightarrow v$ such that $r = u\sigma$ for some substitution $\sigma$, and 2) obtain a new current term $t' = t[v\sigma]_p$ by replacing the subterm $r$ in $t$ by $v\sigma$. In rewrite systems, each step of a reduction can be faced with choices among several candidate redex (reducible

expressions) and rewrite rules. PRS [21], [2] impose an order of priority among rewrite rules and always apply the highest priority rule at each reduction step:

**Definition 4** *A labeled* **PRS** *is a 4-tuple* $(\mathcal{T}(\mathcal{F},\mathcal{H}),\mathcal{L},\mathcal{R},\succ)$ *defining a set* $\mathcal{R}$ *of labeled rewrite rules over a set* $\mathcal{T}(\mathcal{F},\mathcal{H})$ *of terms, where* $\mathcal{L}$ *is a set of labels and* $\succ$ *is a partial (total) order defined among the set* $\mathcal{R}$ *of rewrite rules. (Labels have two purposes: to allow convenient reference to rules and to ease the definition of the order among them.)*

**Example 1.** Let's define the addition over natural numbers as a labeled PRS. To model natural numbers, we use the traditional representation: a functor 0 of arity 0, i.e. $\mathcal{F}_0 = \{0\}$, and two functors $s$ and $p$ that mean successor and predecessor respectively, i.e. $\mathcal{F}_1 = \{p, s\}$. The functor *plus* plays the role of the addition function, i.e. $\mathcal{F}_2 = \{plus\}$. By definition, $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2$. We use two variables, $x$ and $y$, i.e. $\mathcal{H} = \{x, y\}$, and the set of labels $\mathcal{L} = I\!N$ to define the set of rewrite rules $\mathcal{R}$ as:
$\{1 : p(0) \to 0, 2 : p(s(x)) \to x, 3 : plus(x, 0) \to x, 4 : plus(x, y) \to s(plus(x, p(y)))\}$
The priority relation uses the standard order to give priority to rules through their labels. Rule 1 has the highest one, followed by rules 2, 3 and 4. □

The *operational semantics* of reduction in PRS we use here is defined as sequences of rewrite steps called I-reductions [21].

**Definition 5** *A term* $u$ *is* **I-reducible**, *and* **I-rewrites (I-reduces)** *to* $v$ *using a PRS* $\mathcal{PRS}$ *if:*

- $u$ *contains a ground subterm (a redex containing no variables)* $m$ *at position* $p$*, no proper subterm of which is I-reducible;*
- $\mathcal{PRS}$ *contains a rule* $s \to t$*, such that,*
  - $m$ *matches* $s$ *with a matching substitution* $\sigma$ *(i.e.,* $m = s\sigma$*),*
  - $v$ *is the result of replacing* $m$ *in* $u$ *by* $t\sigma$ *(i.e.* $v = u[t\sigma]_p$*), and*
  - $m$ *is not I-reducible by any rule in* $\mathcal{PRS}$ *of higher priority than* $s \to t$*.*

This reduction strategy has two desirable properties: it is decidable and it ensures closeness under context application and substitution with its innermost-first ground reduction strategy. The abbreviation for "$p$ I-rewrites to $q$ using the PRS $\mathcal{P}$" is $p \to^I_{\mathcal{P}} q$. The reflective transitive closure of I-rewriting is noted $\to^{*I}_{\mathcal{P}}$. If $p \to^{*I}_{\mathcal{P}} q$, then we say that $q$ is a *reduct* of $p$.

### 3.2. Mapping program configurations and expressions

Basically, the mapping $\Delta$ takes a program configuration $\mathcal{C}$ and gives a PRS $\mathcal{PRS} = (\mathcal{T}(\mathcal{F},\mathcal{H}),\mathcal{L},\mathcal{R},\succ)$. A message and its current continuation are mapped to a ground term $t \in \mathcal{T}(\mathcal{F},\mathcal{H})$ to be reduced by $\mathcal{PRS}$. A program configuration $\mathcal{C}$ is a set of objects including meta-objects, methods, slot names, continuations or end-user

*Table 1.* Syntactic identities.

| Base language syntax | PRS syntax |
|---|---|
| `'metaObject` | $\mu$ |
| `'lookup` | $\gamma$ |
| `'apply-to` | $\alpha$ |
| `'apply-cont-to` | $\delta$ |
| `'(a1 a2 a3)` | $[a_1, a_2, a_3]$ |

objects, which comprise the running program at a particular point of its execution. To perform the reflective introspection phase, we map part of the information in $\mathcal{C}$ into $\mathcal{PRS}$, the one that is really needed by the rewriting process.

We first show how to represent message execution, since this will govern the way rewrite rules will be laid out concretely. Messages are evaluated in a standard way, using an interpreter `eval` with two arguments: the message and the current continuation[2]. A message has three components: a receiver, a selector and an array of arguments. To mimic the eval function, we represent the message and continuation to be evaluated as a term with functor *eval* with four arguments: a term representing the receiver, one representing the selector, one representing the array of arguments and one that represents the current continuation. Assume that we have a function $\varphi$ from objects to terms, then the message `(p 'selector a1 a2 a3)` with the current continuation object `k` are mapped to the term $eval(rec, sel, [a_1, a_2, a_3], cont)$, where $rec = \varphi(\texttt{p})$, $sel = \varphi(\texttt{'selector})$, $a_1$, $a_2$ and $a_3$ are equal to $\varphi(\texttt{a1})$, $\varphi(\texttt{a2})$ and $\varphi(\texttt{a3})$ respectively, and $cont = \varphi(\texttt{k})$.

Now, we must represent the reflective introspection protocol as a rewrite rule. The protocol makes an intensive use of four selectors: `'metaObject`, `'lookup`, `'apply-to` and `'apply-cont-to`. To make our PRS more compact and easier to read, we map those four selectors to four special functors: $\mu$, $\gamma$, $\alpha$ and $\delta$ respectively. Also, lists of arguments are represented using a standard notation[3] (see Table 1). Consider a simplified formulation of the reflective introspection protocol (§2.3):

```
(o 's a) ⇒ (((o 'metaObject) 'lookup 's o) 'apply-to o '(a) k)
```

The main inconvenience with this formulation is that intermediate continuations in the sequence of messages don't appear explicitly. When we apply the introspection rule to the second message (`lookup`), we need a reification of its own continuation (which is the above apply message). We formulate the rewrite rule in continuation-passing style (CPS) to make intermediate continuations immediately usable. Two continuations are explicitly created by the protocol. The first one waits for a meta-object to which it sends a lookup message and the second waits for a method object to which it sends an apply message. We will capture these as terms of the form $c(sel, args, cont)$. Invoking them on an object $o$ sends $o$ the message with selector *sel* and arguments *args* whose result will be passed to the continuation *cont*. This leads to the following reformulation for the introspection protocol rewrite rule, which is given the lowest priority in our PRS:

*Table 2.* Basic priority rewrite system.

| | | |
|---|---|---|
| $\mathcal{F}_0$ | = | $\{BMO, BA, BL, ROOT, BAC, IK, \mu, \gamma, \alpha, \delta, nil, doesNotUnderstand, noSuchObject\}$ |
| $\mathcal{F}_2$ | = | $\{cons, blf\}$ $\mathcal{F}_3 = \{c\}$ $\mathcal{F}_4 = \{eval, baf\}$ |
| $\mathcal{F}$ | = | $\mathcal{F}_0 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4$ |
| $\mathcal{H}$ | = | $\{M, O, A, R, S, K, K', K''\}$ |
| $\mathcal{L}$ | = | $\{h, m, l\}$ |
| $\succ$ | = | $\{(h, m), (h, l), (m, l)\}$ |
| $\mathcal{R}$ | = | *see below* |

| | |
|---|---|
| $h : eval(BMO, \mu, [], K) \rightarrow eval(K, \delta, [BMO], IK)$ | (1) |
| $h : eval(BL, \mu, [], K) \rightarrow eval(K, \delta, [BMO], IK)$ | (2) |
| $h : eval(BA, \mu, [], K) \rightarrow eval(K, \delta, [BMO], IK)$ | (3) |
| $h : eval(c(\gamma, A, K), \delta, [R], K') \rightarrow eval(R, \gamma, A, K)$ | (4) |
| $h : eval(c(\alpha, [O, A, K], K'), \delta, [R], K'') \rightarrow eval(R, \alpha, [O, A, K], K')$ | (5) |
| $h : eval(BMO, \gamma, [\gamma, BMO], K) \rightarrow eval(K, \delta, [BL], IK)$ | (6) |
| $h : eval(BMO, \gamma, [\alpha, BL], K) \rightarrow eval(K, \delta, [BA], IK)$ | (7) |
| $h : eval(BMO, \gamma, [\alpha, BA], K) \rightarrow eval(K, \delta, [BA], IK)$ | (8) |
| $m : eval(BA, \alpha, [M, [O, A, K], K'], K'') \rightarrow baf(M, O, A, K)$ | (9) |
| $h : eval(BA, \alpha, [BL, [BMO, [S, P], K], K'], K'') \rightarrow eval(K, \delta, [blf(S, P)], IK)$ | (10) |
| $l : blf(S, P) \rightarrow doesNotUnderstand$ | (11) |
| $h : eval(doesNotUnderstand, S, R, K) \rightarrow doesNotUnderstand$ | (12) |
| $m : eval(O, \mu, [], K) \rightarrow noSuchObject$ | (13) |
| $l : eval(O, S, A, K) \rightarrow eval(O, \mu, [], c(\gamma, [S, O], c(\alpha, [O, A, K], IK)))$ | (14) |

$$eval(O, S, A, K) \rightarrow eval(O, \mu, [], c(\gamma, [S, O], c(\alpha, [O, A, K], IK)))$$

Now, we are faced with our first infinite meta-regression. The right-hand side of this rewrite rule can be reduced again using the same rule. Informally, it means that if we need the meta-object $mo$ of $o$ to send the message, we must not need $mo$ to find $mo$. This forces us to adopt our first important assumption: meta-objects can be fetched primitively without using the reflective introspection protocol. In the PRS, for each object `o` in the program configuration $\mathcal{C}$, we will get a meta-object fetching rule of the form $eval(o, \mu, [], K) \rightarrow eval(K, \delta, [mo], IK)$ where $o = \varphi(\texttt{o})$ and $mo = \varphi(\texttt{mo})$, the meta-object of `o`. In our PRS, these rules will be given the highest priority. For the kernel objects of Fig 1, this gives the rewrite rules (1-3) in Table 2. Because the invocation of the continuation $K$ will discard the current continuation, for simplicity we chose this one to be $IK$, the identity continuation.

Now, we must deal with terms representing the application of continuations, such as exhibited in the right-hand sides of meta-object fetching rewrite rules. Blindly using the reflective introspection rewrite rule can lead to another meta-regression since it introduces more continuations to apply existing ones. Hence, we introduce the rules (4) and (5), having higher priority than the reflective rule, to apply the continuations constructed during the reduction.

If we concentrate on the kernel as defined in §2 and illustrated in Fig.1, some other potential infinite meta-regressions must be prevented in order to make the reflective

protocol work properly. Recall that applying the reflective rewrite rule calls for sending three messages. We have dealt with the meta-object fetching messages. Now, we must examine lookup and apply messages. Infinite meta-regression may appear when we send a lookup message to the basic meta-object $BMO$ in order to find its lookup method. Using the reflective rewrite rule would lead to:

$$\begin{aligned} &eval(BMO, \gamma, [\gamma, BMO], K) \rightarrow \\ &\quad eval(BMO, \mu, [], c(\gamma, [\gamma, BMO], c(\alpha, [BMO, [\gamma, BMO], K], IK))) \end{aligned}$$

$BMO$ is its own metaobject, so the next reduction steps (rules 1 and 4) would send exactly the same lookup message to $BMO$. We introduce the rules (6-8) to avoid this as well as two similar problems to find the apply methods of $BL$ and $BA$.

The last problem we have to fix is how to relate actions such as sending to the object $BA$ an apply message asking for the execution of some method $m$ on an object $o$ with arguments $a_1, \ldots, a_n$ and continuation $k$, to the rock-bottom implementation. What we need is a way to jump from the language level of message sending and reflective protocol to the realm of basic functions that actually implement the language. In addition to the functions introduced in §2, we rely on the function *baf*(`m`,`o`,`a`,`k`), which executes `m` on `o` with arguments `a` and continuation `k`. A call to this function is represented in the PRS as a term $baf(m, o, a, k)$ and the jump from message sending to the implementation introduces the rule (9) having higher priority than the reflective rule.

But we also have another basic function to perform the lookup for objects having $BMO$ as their meta-object. Instead of using *baf* to execute `BL`, we will take advantage of the function *blf*. This gives the rule (10) having a higher priority than (9) to bypass it when the method to be executed is $BL$. We will assume for now that there will be rules to define $blf$ in the PRS, but to catch errors, we add the rule (11) with the lowest priority. Unfortunately, this rule may lead to situations where we will try to send lookup messages to *doesNotUnderstand* (apply rule (10) then (4)). We assume in this case that the rewriting process will stop on the term *doesNotUnderstand*, a decision which is implemented by the rule (12). By the same token, we add the rule (13) to catch the errors when messages are sent to non-existing objects. This rule mimics a meta-object fetching rule but it has lower priority than these; hence, it will be fired only when the receiver has no meta-object and this will only happen when the receiver term does not represent an existing object.

This completes the set of rewrite rules which are necessary to model the essential features of the kernel. We need three different priorities to cope with the previous requirements. The Table 2 uses labels $l, m, h$ to suggest lower, medium and higher priority respectively. We now give a simple example of the rewriting process where the current program configuration contains an object `o` that has a method `m`, which only uses the standard behavior implemented by the kernel.

*Table 3.* Complete reduction for the example 2. Each rewriting step is labeled with the rule number used to rewrite the preceding term (to shorten the process and to make it easier to read we have combined the steps consisting only of applying continuations with the following one whenever it was possible). The first part (before the horizontal line) is the lookup phase while the second part is the apply phase.

| Rewriting steps | Rules |
|---|---|
| $eval(o, s, [a], k)$ | |
| $\rightarrow eval(o, \mu, [], c(\gamma, [s, o], c(\alpha, [o, [a], k], IK)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [s, o], c(\alpha, [o, [a], k], IK))$ | (16+4) |
| $\rightarrow eval(BMO, \mu, [], c(\gamma, [\gamma, BMO], c(\alpha, [BMO, [s, o], c(\alpha, [o, [a], k], IK)], IK)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\gamma, BMO], c(\alpha, [BMO, [s, o], c(\alpha, [o, [a], k], IK)], IK))$ | (1+4) |
| $\rightarrow eval(BL, \alpha, [BMO, [s, o], c(\alpha, [o, [a], k], IK)], IK)$ | (6+5) |
| $\rightarrow eval(BL, \mu, [], c(\gamma, [\alpha, BL], c(\alpha, [BL, [BMO, [s, o], c(\alpha, [o, [a], k], IK)], IK], IK)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\alpha, BL], c(\alpha, [BL, [BMO, [s, o], c(\alpha, [o, [a], k], IK)], IK], IK))$ | (2+4) |
| $\rightarrow eval(BA, \alpha, [BL, [BMO, [s, o], c(\alpha, [o, [a], k], IK)], IK], IK)$ | (7+5) |
| $\rightarrow eval(c(\alpha, [o, [a], k], IK), \delta, [blf(s, o)], IK)$ | (10) |
| $\rightarrow eval(c(\alpha, [o, [a], k], IK), \delta, [m], IK)$ | (18) |
| $\rightarrow eval(m, \alpha, [o, [a], k], IK)$ | (5) |
| $\rightarrow eval(m, \mu, [], c(\gamma, [\alpha, m], c(\alpha, [m, [o, [a], k], IK], IK)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\alpha, m], c(\alpha, [m, [o, [a], k], IK], IK))$ | (16+4) |
| $\rightarrow eval(BMO, \mu, [], c(\gamma, [\gamma, BMO], c(\alpha, [BMO, [\alpha, m],$ $c(\alpha, [m, [o, [a], k], IK], IK)], IK)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\gamma, BMO], c(\alpha, [BMO, [\alpha, m], c(\alpha, [m, [o, [a], k], IK], IK)], IK))$ | (1+4) |
| $\rightarrow eval(BL, \alpha, [BMO, [\alpha, m], c(\alpha, [m, [o, [a], k], IK], IK)], IK)$ | (6+5) |
| $\rightarrow eval(BL, \mu, [], c(\gamma, [\alpha, BL], c(\alpha, [BL, [BMO, [\alpha, m],$ $c(\alpha, [m, [o, [a], k], IK], IK)], IK], IK)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\alpha, BL], c(\alpha, [BL, [BMO, [\alpha, m],$ $c(\alpha, [m, [o, [a], k], IK], IK)], IK], IK))$ | (2+4) |
| $\rightarrow eval(BA, \alpha, [BL, [BMO, [\alpha, m], c(\alpha, [m, [o, [a], k], IK], IK)], IK], IK)$ | (7+5) |
| $\rightarrow eval(c(\alpha, [m, [o, [a], k], IK], IK), \delta, [blf(\alpha, m)], IK)$ | (10) |
| $\rightarrow eval(BA, \alpha, [m, [o, [a], k], IK], IK)$ | (17+5) |
| $\rightarrow baf(m, o, [a], k)$ | (9) |

**Example 2.** Consider a program configuration consisting of the kernel objects plus an object `o` with a slot named `'s` pointing to a method object `m` of one argument. Assume that `BMO` is the meta-object of `o` and `BA`, the apply method of `m`. This configuration can be mapped to a PRS where the set of rewrite rules contains the fourteen rules for the kernel plus the four rules below describing the two new objects, where $o = \varphi(\texttt{o})$, $m = \varphi(\texttt{m})$ and $s = \varphi(\texttt{'s})$.

| | |
|---|---|
| $h : eval(o, \mu, [], K) \rightarrow eval(K, \delta, [BMO], IK)$ | (15) |
| $h : eval(m, \mu, [], K) \rightarrow eval(K, \delta, [BMO], IK)$ | (16) |
| $h : blf(\alpha, m) \rightarrow BA$ | (17) |
| $h : blf(s, o) \rightarrow m$ | (18) |

The Table 3 illustrates the complete reduction process for a message `(o 's a1)` and current continuation `k`. The evaluation of the message is represented as the term $eval(o, s, [a], k)$ where $a = \varphi(\texttt{a1})$ and $k = \varphi(\texttt{k})$. □

### 3.3. Formal semantics and termination

#### *3.3.1. Sound and complete priority rewrite systems*

By itself, the operational semantics of PRS, as defined in Definition 5, doesn't ensure termination of reductions. The formal semantics [21] of a PRS is defined by mapping it to the logical theory (a set of equational first-order formulas) that characterizes it and whose logical consequences are computed via the PRS. A PRS is then *sound* iff all the reductions it computes are logical consequences of the corresponding theory and *complete* iff all logical consequences of the theory correspond to reductions in the PRS. The necessary conditions to ensure the soundness and completeness of PRS are based upon the notion of definitional PRS defined as follows [21]:

**Definition 6** *Given a PRS $\mathcal{PRS}$, let's partition the set $\mathcal{F}$ of functors (function symbols) into defined functions, i.e. whose functors appears as outermost symbol in left-hand sides of rewrite rules, and constructor ones. A constructor term is one where no functors corresponding to defined functions appear. $\mathcal{PRS}$ is a* **definitional priority rewrite system (DPRS)** *if the left-hand side of each rule is of the form "$f(\tau)$", where $f$ corresponds to a defined function (by definition) and $\tau$ is a tuple of constructor terms.*

The soundness and completeness of a PRS ensure that *each ground term has a unique constructor term reduct.* They are shown in three parts: termination of ground reduction (reduction limited to ground subterms, see 3.1), ground confluence and reducibility of every non-ground constructor term. Termination means that there is no infinite chain of rewrites. Proving termination is one of the hardest problems in rewriting systems.

The following proposition gives conditions under which ground confluence (the second requirement) holds for DPRS ([21], proposition 4):

**Proposition 1** *A DPRS is* **ground confluent** *if the following conditions are satisfied: for every pair of rules $r_1 : f(\tau_1) \rightarrow t_1$ and $r_2 : f(\tau_2) \rightarrow t_2$ whose left-hand sides unify with most general unifier $\sigma$, one of the following holds:*

- *one of the rules $r_1$ or $r_2$ has a higher priority that the other;*
- *$f(\tau_1)\sigma$ is an instance of the left-hand side of another rule of higher priority than $r_1$ (or $r_2$);*
- *$t_1\sigma$ and $t_2\sigma$ are identical.*

The final requirement is that every ground non-constructor term must be reducible. The easiest way to achieve this condition is to provide catch-all rewrite rules for all defined functions (i.e. for all functors $f$ of defined functions, provide a rule whose left-hand side is of the form $f(x_1,\ldots,x_n)$ where $x_1$, ..., $x_n$ are variables). Such a strategy is easy to implement without disturbing the other requirements simply by assigning those catch-all rules the lowest priority.

*3.3.2. Mapping program configurations to sound and complete PRS*

Program configurations in our reflective language can be mapped to sound and complete PRS. But before proving this assertion, we must define the mapping $\Delta$ from program configurations to PRS. First, we represent objects and slot names as terms in the PRS. Basically, a program configuration is a set $\mathcal{C}$ of objects existing in the program at a certain point in its execution. To manipulate these objects within the PRS, we map them into *0-arity* terms.

**Proposition 2** *For any program configuration $\mathcal{C}$, there exists a set $\overline{\mathcal{F}}$ of 0-arity terms and a function $\varphi : \mathcal{C} \rightarrow \overline{\mathcal{F}}$ such that $\mathcal{C}$ and $\overline{\mathcal{F}}$ have equal cardinality, i.e. $\#\mathcal{C} = \#\overline{\mathcal{F}}$, and for all $c \in \mathcal{C}$, there is one and only one $f \in \overline{\mathcal{F}}$ such that $\varphi(c) = f$.*

**Proof:** Straightforward by construction. The set $\overline{\mathcal{F}}$ can be any set of 0-arity functors whose cardinality equals the one of $\mathcal{C}$. $\varphi$ is then defined as a one to one mapping from $\mathcal{C}$ to $\overline{\mathcal{F}}$. ■

For readability, it is more convenient to assume that $\overline{\mathcal{F}}$ is constructed in a more disciplined way. We distinguish two subsets in $\mathcal{C}$: $\mathcal{S}$ is the set of objects representing slot names and $\mathcal{O} = \mathcal{C} \backslash \mathcal{S}$ contains the remaining objects. $\overline{\mathcal{F}}$ should first contain the predefined functors used as the set $\mathcal{F}_0$ in Table 2. For all other objects $o \in \mathcal{O}$ and $s \in \mathcal{S}$ not represented in $\mathcal{F}_0$, $\overline{\mathcal{F}}$ can be completed with functors such as $o1$, $o2$, ... and $s1$, $s2$, ... respectively. $\overline{\mathcal{F}}$ is then the set of 0-arity functors in the set $\mathcal{T}(\mathcal{F}, \mathcal{H})$ of first-order terms in our PRS. $\varphi$ maps `BMO`, `BL`, `BA`, etc. to the functors $BMO$, $BL$, $BA$, ..., as well as selectors (slot names) `'metaObject`, `'lookup`, etc. to the functors $\mu$, $\gamma$, ... Otherwise, it simply maps objects and selectors to terms of the form $o\,n$ and $s\,m$, $n, m \geq 0$.

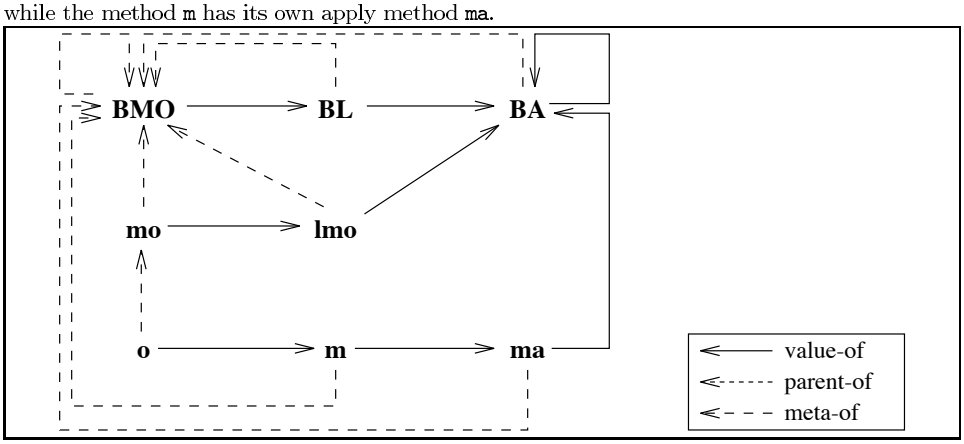
**Definition 7** *For any program configuration $\mathcal{C}$, given:*

- *a set $\overline{\mathcal{F}}$ of 0-arity functors such that $\#\overline{\mathcal{F}} = \#\mathcal{C}$),*
- *a set $\mathcal{T}(\mathcal{F}, \mathcal{H})$ of first-order terms defined by the following: $\mathcal{F}_0 = \overline{\mathcal{F}}$, $\mathcal{F}_2 = \{eval, cons, blf\}$, $\mathcal{F}_3 = \{c\}$, $\mathcal{F}_4 = \{baf\}$, $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \cup \mathcal{F}_4$ and $\mathcal{H} = \{M, O, A, R, S, K, K', K''\}$,*
- *a function $\varphi : \mathcal{C} \rightarrow \mathcal{F}_0 \subset \mathcal{T}(\mathcal{F}, \mathcal{H})$*
- *and a set of labels $\mathcal{L} = \{h, m, l\}$ with an order relation $\gg = \{(h, m), (h, l), (m, l)\}$.*

*we define the* `mapping` *$\Delta : \mathcal{C} \rightarrow (\mathcal{T}(\mathcal{F}, \mathcal{H}), \mathcal{L}, \mathcal{R}, \succ)$ where $\succ$ is isomorphic to $\gg$, and $\mathcal{R}$ contains the rules obtained by the following:*

1. *For any* `o` $\in \mathcal{C}$ *with meta-object* `mo`*, add a rule: $h : eval(o, \mu, [], K) \rightarrow eval(K, \delta, [mo], IK)$, where $o = \varphi($`o`$)$ and $mo = \varphi($`mo`$)$.*
2. *For any* `o` $\in \mathcal{O}$ *whose meta-object is* `BMO` *and for all selectors* `s` $\in \mathcal{S}$ *such that blf(*`s`*,*`o`*)* $\Rightarrow$ `method`*, add a rule: $h : blf(s, o) \rightarrow m$, where $o = \varphi($`o`$)$, $s = \varphi($`s`$)$ and $m = \varphi($`method`$)$*
3. *Add the rewrite rules (4) to (14) already defined in Table 2.*

*Figure 2.* Program configuration (Example 3): and object `o` has a specialized lookup method `lmo` while the method `m` has its own apply method `ma`.




The PRS obtained by the above mapping is consistent with the one presented in §3.2. The mapping $\Delta$ generates slightly more rules since it will generate a meta-object fetching rule for all objects in the configuration (in $\mathcal{O}$) and many rules with left-hand sides of the form $blf(s, o)$ for the kernel objects, which were not necessary in the example. But these additional rules do not change the fundamental properties of the resulting PRS. Using the PRS, a message is mapped to a term which is then used as a query (to reduce it by rewriting):

**Definition 8** *A message with receiver* $\mathtt{p} \in \mathcal{C}$*, selector* $\mathtt{s} \in \mathcal{S}$*, arguments* $\mathtt{a1}, \ldots, \mathtt{an} \in \mathcal{C}$*, and current continuation object* $\mathtt{k} \in \mathcal{O}$ *is mapped to the term* $t = eval(rec, sel, [a_1, \ldots, a_n], cont)$ $(\in \mathcal{T}(\mathcal{F}, \mathcal{H})$ *as defined in Def. 7) where* $rec = \varphi(\mathtt{p})$*,* $sel = \varphi(\mathtt{s})$*,* $a_1 = \varphi(\mathtt{a1})$*, ...,* $a_n = \varphi(\mathtt{an})$ *and* $cont = \varphi(\mathtt{k})$*.*

**Proposition 3** *For any program configuration* $\mathcal{C}$ *which respects the principles* `P1` *to* `P12`*, the PRS* $\Delta(\mathcal{C})$ *is* **sound** *and* **complete.**

**Proof:** Follows immediately from propositions 4, 5 and 6 in the appendix ■

## 4. Discussion

Interesting outcomes of characterizing our reflective protocol as priority rewrite systems lie in a better understanding of reflection and of its implementation.

### 4.1. General object behavior

In general, an object `o` has $n$ levels of meta-objects, i.e. `o` has a meta-object `mo`, which itself has a meta-meta-object, and so on until `BMO` is reached. As requested

by the reflective protocol, each of these meta-objects has a lookup method. The purpose of creating such a hierarchy of meta-objects is to have specialized lookup methods. Similarly, a method object `m` has its own apply method `ma`, which itself has its own apply method and so on until `BA` is reached. To make things more concrete, consider the following example where an object has its own specialized lookup and where the method to be executed has its own apply method.

**Example 3 (Part I).** Consider the program configuration depicted in Figure 2. The following table gives the rewrite rules added to the one of Table 2 by $\Delta$ (we omit certain rules which are not used for the moment).

| | |
|---|---|
| $h : eval(o, \mu, [], K) \rightarrow eval(K, \delta, [mo], ik)$ | (19) |
| $h : eval(mo, \mu, [], K) \rightarrow eval(K, \delta, [BMO], ik)$ | (20) |
| $h : eval(lmo, \mu, [], K) \rightarrow eval(K, \delta, [BMO], ik)$ | (21) |
| $h : blf(\gamma, mo) \rightarrow lmo$ | (22) |
| $h : blf(\alpha, lmo) \rightarrow BA$ | (23) |

The Table 4 illustrates the reduction process for the lookup phase assuming a message `(o 's a)` and current continuation `k`. □

The lookup phase can be split in two major parts and several sub-parts. In the first part (Table 4, above the twin lines), the reduction process finds the lookup method associated with $mo$. This is done in three subparts. First, it dives into the meta-object hierarchy until it finds $BMO$. Next, it retrieves $BL$, the lookup method of $BMO$. And finally, it applies $BL$ to $mo$ in order to find its lookup method $lmo$. The second part (under the twin horizontal lines) applies $lmo$ to $o$ in order to find the method corresponding to the selector $s$. This is done in four subparts: it dives in the meta-object hierarchy of $lmo$ until $BMO$, it finds the lookup method $BL$ of $BMO$, it applies $BL$ to $lmo$ to find its apply method $BA$ and, finally, it applies $BA$ to $lmo$.

This ends with the reduct $baf(lmo, mo, [s, o], c(\alpha, [o, [a], k], ik))$. Contrary to Example 2, the rewrite system ignores the semantics of $lmo$. In Example 2, the rewrite system was able to find the appropriate method to respond to the message because the semantics of $BL$ is known to it. Here, the reduction process ends with a term interpreted as a request to the language processor to find the method associated with selector $s$ in $o$ using the lookup method $lmo$. To do that, the reduct is transformed back into a call to the function *baf*.

Hence, we must perform the inverse of the mapping described in Definition 8, a task for which we assume that we have a function $\varphi^{-1}$ from terms to objects. The only problem is to transform the continuation $c(\alpha, [o, [a], k], ik))$ built during the reduction into a continuation object. Hold on to this problem for the moment, we will come back to it shortly. Assume we transform the term into a call to *baf*. Applying $lmo$ finds the method $m$ which is then passed to the continuation $c(\alpha, [o, [a], k], ik))$ by sending it the message with selector $\delta$ (`'apply-cont-to`) and with argument $m$. This message invokes again the rewriting system, as it is now described:

*Table 4.* Complete reduction for the example 3 (lookup phase).

| Rewriting steps | Rules |
|---|---|
| $eval(o, s, [a], k)$ | |
| $\rightarrow eval(o, \mu, [], c(\gamma, [s, o], c(\alpha, [o, [a], k], ik)))$ | (14) |
| $\rightarrow eval(mo, \gamma, [s, o], c(\alpha, [o, [a], k], ik))$ | (19+4) |
| $\rightarrow eval(mo, \mu, [], c(\gamma, [\gamma, mo], c(\alpha, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\gamma, mo], c(\alpha, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik))$ | (20+4) |
| $\rightarrow eval(BMO, \mu, [], c(\gamma, [\gamma, BMO], c(\alpha, [BMO, [\gamma, mo], c(\alpha, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik)], ik)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\gamma, BMO], c(\alpha, [BMO, [\gamma, mo], c(\alpha, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik)], ik))$ | (1+4) |
| $\rightarrow eval(BL, \alpha, [BMO, [\gamma, mo], c(\alpha, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik)], ik)$ | (6+5) |
| $\rightarrow eval(BL, \mu, [], c(\gamma, [\alpha, BL], c(\alpha, [BL, [BMO, [\gamma, mo], c(\alpha, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik)], ik], ik)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\alpha, BL], c(\alpha, [BL, [BMO, [\gamma, mo], c(\alpha, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik)], ik], ik))$ | (2+4) |
| $\rightarrow eval(BA, \alpha, [BL, [BMO, [\gamma, mo], c(\alpha, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik)], ik], ik)$ | (7+5) |
| $\rightarrow eval(c(\alpha, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik), \delta, [blf(\gamma, mo)], ik)$ | (10) |
| $\rightarrow eval(lmo, \alpha, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik)$ | (23+5) |
| $\rightarrow eval(lmo, \mu, [], c(\gamma, [\alpha, lmo], c(\alpha, [lmo, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik], ik)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\alpha, lmo], c(\alpha, [lmo, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik], ik))$ | (21+4) |
| $\rightarrow eval(BMO, \mu, [], c(\gamma, [\gamma, BMO], c(\alpha, [BMO, [\alpha, lmo], c(\alpha, [lmo, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik], ik)], ik)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\gamma, BMO], c(\alpha, [BMO, [\alpha, lmo], c(\alpha, [lmo, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik], ik)], ik))$ | (1+4) |
| $\rightarrow eval(BL, \alpha, [BMO, [\alpha, lmo], c(\alpha, [lmo, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik], ik)], ik)$ | (6+5) |
| $\rightarrow eval(BL, \mu, [], c(\gamma, [\alpha, BL], c(\alpha, [BL, [BMO, [\alpha, lmo], c(\alpha, [lmo, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik], ik)], ik], ik)))$ | (14) |
| $\rightarrow eval(BMO, \gamma, [\alpha, BL], c(\alpha, [BL, [BMO, [\alpha, lmo], c(\alpha, [lmo, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik], ik)], ik], ik))$ | (2+4) |
| $\rightarrow eval(BA, \alpha, [BL, [BMO, [\alpha, lmo], c(\alpha, [lmo, [mo, [s, o],$ $c(\alpha, [o, [a], k], ik)], ik], ik)], ik], ik)$ | (7+5) |
| $\rightarrow eval(c(\alpha, [lmo, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik], ik), \delta, [blf(\alpha, lmo)], ik)$ | (10) |
| $\rightarrow eval(BA, \alpha, [lmo, [mo, [s, o], c(\alpha, [o, [a], k], ik)], ik], ik)$ | (24+5) |
| $\rightarrow baf(lmo, mo, [s, o], c(\alpha, [o, [a], k], ik))$ | (9) |

**Example 3 (Part II).** Consider again the program configuration depicted in Figure 2. The rewrite rules added by the mapping $\Delta$ for the application of the method $m$ (we still omit rules not used in the reduction) are:

| | |
|---|---|
| $h : eval(m, \mu, [], K) \rightarrow eval(K, \delta, [BMO], ik)$ | (24) |
| $h : eval(ma, \mu, [], K) \rightarrow eval(K, \delta, [BMO], ik)$ | (25) |
| $h : blf(s, o) \rightarrow m$ | (26) |
| $h : blf(\alpha, m) \rightarrow ma$ | (27) |
| $h : blf(\alpha, ma) \rightarrow BA$ | (28) |

*Table 5.* Complete reduction for the example 3 (apply phase).

| Rewriting steps | Rules |
|---|---|
| $eval(c(\alpha,[o,[a],k],ik),\delta,[m],ik)$ | |
| $\rightarrow eval(m,\alpha,[o,[a],k],ik)$ | (5) |
| $\rightarrow eval(m,\mu,[],c(\gamma,[\alpha,m],c(\alpha,[m,[o,[a],k],ik],ik)))$ | (14) |
| $\rightarrow eval(BMO,\gamma,[\alpha,m],c(\alpha,[m,[o,[a],k],ik],ik))$ | (24+4) |
| $\rightarrow eval(BMO,\mu,[],c(\gamma,[\gamma,BMO],c(\alpha,[BMO,[\alpha,m],c(\alpha,[m,[o,[a],k],ik],ik)],ik)))$ | (14) |
| $\rightarrow eval(BMO,\gamma,[\gamma,BMO],c(\alpha,[BMO,[\alpha,m],c(\alpha,[m,[o,[a],k],ik],ik)],ik))$ | (1+4) |
| $\rightarrow eval(BL,\alpha,[BMO,[\alpha,m],c(\alpha,[m,[o,[a],k],ik],ik)],ik)$ | (6+5) |
| $\rightarrow eval(BL,\mu,[],c(\gamma,[\alpha,BL],c(\alpha,[BL,[BMO,[\alpha,m],$ $c(\alpha,[m,[o,[a],k],ik],ik)],ik],ik)))$ | (14) |
| $\rightarrow eval(BMO,\gamma,[\alpha,BL],c(\alpha,[BL,[BMO,[\alpha,m],c(\alpha,[m,[o,[a],k],ik],ik)],ik],ik))$ | (2+4) |
| $\rightarrow eval(BA,\alpha,[BL,[BMO,[\alpha,m],c(\alpha,[m,[o,[a],k],ik],ik)],ik],ik)$ | (7+5) |
| $\rightarrow eval(c(\alpha,[m,[o,[a],k],ik],ik),\delta,[blf(\alpha,m)],ik)$ | (10) |
| $\rightarrow eval(ma,\alpha,[m,[o,[a],k],ik],ik)$ | (27+5) |
| $\rightarrow eval(ma,\mu,[],c(\gamma,[\alpha,ma],c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik)))$ | (14) |
| $\rightarrow eval(BMO,\gamma,[\alpha,ma],c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik))$ | (25+4) |
| $\rightarrow eval(BMO,\mu,[],c(\gamma,[\gamma,BMO],c(\alpha,[BMO,[\alpha,ma],$ $c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik)],ik)))$ | (14) |
| $\rightarrow eval(BMO,\gamma,[\gamma,BMO],c(\alpha,[BMO,[\alpha,ma],c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik)],ik))$ | (1+4) |
| $\rightarrow eval(BL,\alpha,[BMO,[\alpha,ma],c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik)],ik)$ | (6+5) |
| $\rightarrow eval(BL,\mu,[],c(\gamma,[\alpha,BL],c(\alpha,[BL,[BMO,[\alpha,ma],$ $c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik)],ik],ik)))$ | (14) |
| $\rightarrow eval(BMO,\gamma,[\alpha,BL],c(\alpha,[BL,[BMO,[\alpha,ma],$ $c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik)],ik],ik))$ | (2+4) |
| $\rightarrow eval(BA,\alpha,[BL,[BMO,[\alpha,ma],c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik)],ik],ik)$ | (7+5) |
| $\rightarrow eval(c(\alpha,[ma,[m,[o,[a],k],ik],ik],ik),\delta,[blf(\alpha,ma)],ik)$ | (10) |
| $\rightarrow eval(BA,\alpha,[ma,[m,[o,[a],k],ik],ik],ik)$ | (28+5) |
| $\rightarrow baf(ma,m,[o,[a],k],ik)$ | (9) |

The Table 5 illustrates the reduction process after the lookup phase has found the method $m$ associated with selector $s$ in $o$. □

The apply phase can be split in three parts. In the first part, the rewrite system finds $ma$, the apply method of $m$, a task that launches a short (because the meta-object of $m$ is $BMO$) lookup phase. The second part finds the apply method of $ma$, which turns out to be $BA$. The last part applies $BA$ to $ma$ and ends with a term $baf(ma,m,[o,[a],k],ik)$, which again represents a call to the basic apply function. To execute the rest of the program, this term is transformed back into a call to *baf*, as we discussed after the lookup phase.

The Example 3 just gives a glimpse of the complexity arising from a systematic use of reflection. For instance, assume the lookup method *lmo* does not have $BA$ as its apply method but another one called *lmoApp*. If the apply method of *lmoApp* is $BA$, then the result of the reduction process yields the following term:

$$baf(lmoApp,lmo,[mo,[s,o],c(\alpha,[o,[a],k],ik)],ik)$$

This term represents the execution of the method *lmoApp* on *lmo* which is itself a method. *lmoApp* acts as a local interpreter for *lmo* which is executed on *mo* to find the method associated with selector $s$ in $o$. The same kind of behavior may happen during the apply phase where, in general, we could have several more levels of apply methods than in Example 3. Assume, for instance, that the apply method $ma$ has its own apply method $ma^2$ and so on until an apply method $ma^n$ is reached whose apply method is $BA$. The result of a reduction process would then be:

$$baf(ma^n, ma^{n-1}, [\ldots, [ma, [m, [o, [a], k], ik], ik], \ldots], ik)$$

This reduct requires the execution of *baf*, which applies the method $ma^n$ to the object $ma^{n-1}$. But, $ma^n$ is an apply method that executes the method $ma^{n-1}$, which itself executes the method $ma^{n-2}$, and so on until the method $m$ is executed on $o$. In the field of reflection, we recognize that we are in the presence of a *reflective tower* such as the one presented by Smith [24]: a tower of evaluators where the *nth* level evaluator executes the *(n-1)th* level evaluator and so on. In 3-Lisp, an expression of the current tower similar to the above (we'll compare these to ours in the next section) would appear as:

$$\ldots(reduce\,'reduce\,'(\ldots\,'(reduce\ m\ \ '(a)\ e_n\ k_n)\ \ldots)\ e_0\ k_0)\ldots$$

where $e_0, \ldots, e_n$ and $k_0, \ldots, k_n$ are respectively the environments and the continuations of each level. 3-Lisp's towers are theoretically infinite, but when the behavior of a level doesn't deviate from the standard one, this level and all the levels above it can be eliminated and replaced by a basic evaluator. Also, a good implementation of 3-Lisp [7] detects and eliminates as soon as possible a level becoming flat during the execution. Hence, in theory, 3-Lisp should rarely execute a tower of more than two levels. Notice as mentioned earlier the similar flexibility between 3-Lisp and our protocol obtained by the use of explicit continuations.

### 4.2. Efficiency Concerns

A major concern about reflective languages is efficiency. We now briefly come back to the lookup and apply phases to identify the potential sources of inefficiency and to discuss their implementation. The reflective introspection rule is at the heart of our language but should the language processor use it blindly? Gains in efficiency generally call for compiling message sending and applying optimizations in much the same way as Kiczales suggested in [14].

For the lookup phase, the computation is essentially additive, a lookup method is found which itself is used to find the next one and so on. Hence, it can be rather easily optimized. In practice, the hierarchy of meta-objects of an object may not change very often. Standard caching techniques can be used to keep around the current lookup method applying to o. We just need to compute lookup methods associated with each object upon reception of their first message and when changes

are made to their meta-object hierarchy. Hence, the resulting efficiency should equal the one of current object-oriented languages.

For the apply phase, things are a little more complicated since we have this reflective tower of apply methods, one executing the next one below until the apply method `ma` executes `m`. This process is essentially multiplicative, and thus harder to implement efficiently. Actually, each of these apply methods acts as a language processor that defines the semantics of a (new) language in which the method under it is written. Such language processors can be defined either as compilers or interpreters. In the first case, the method $ma^{i-1}$ would be compiled using the "compiling" apply method $ma^i$, into code that can again be compiled by the "compiling" apply method $ma^{i+1}$. Applying this process $n$ times would compile the method $m$ into code interpreted by the language processor directly (*baf*), which could be kept within the method object itself. (This kind of behavior is implied by the specification of the function `make-method-lambda` in the CLOS MOP [15]). The main problem we face with this approach is how to provide end-users with a suitable abstract way of specifying "compiling" apply methods, although users driven by performance should be interested in such an approach even if it looks more complicated. Premises of this approach appear in Lamping et al. [16].

An alternative, and more traditional, approach would be to have "interpreting" apply methods, which would essentially be variants of the basic meta-interpreter of the core language. This approach is well-known to be very inefficient if naively implemented. To obtain the level of efficiency expected from compiling apply methods, some sort of background automatic compilation would be needed. Techniques such as partial evaluation and semantics-based program transformations should play a role, but they are not well mastered yet, especially in object-oriented and imperative languages. In reflection, their use is still in its infancy. Should it be possible to use them, caching techniques could track the result of these transformations and of the optimized code from one application of a method to the other.

### 4.3. The status of the rewriting system

In the actual execution of a program, each message sending triggers a rewriting process, which in the best case (the meta-object of the receiver is `BMO` and the apply method used by the method to be applied is `BA`) yields a term representing the call to be made to the basic apply function to execute the method corresponding to the message. As we have seen in the last example, the rewriting process can stop short of this ideal situation if the receiver has its own specialized lookup method. In this case, rewriting cannot proceed further and it needs to go back to the program in order to execute the lookup method.

Beyond implementation concerns, a striking fundamental question at this point is the status of these rewriting systems versus the current object world. Until now, we have used them to prove the correctness of the *lookup* ∘ *apply* protocol and have suggested that they must be part of the execution of programs. In fact, we can envision three possible implementations, each of which should be investigated since

at present none appears to have an edge over the others: (1) absorb the rewriting system within the evaluation process, (2) absorb the evaluation and application processes into the rewriting system, or (3) keep the evaluation process and the rewriting system independent of each other.

The first two alternatives simplify the coordination between method execution and rewriting by unifying them into a single framework. Because the rewriting system is composed of a fixed set of rewriting rules, except for very simple rules that encode meta-object fetching in general and the lookup process for objects that have `BMO` as their meta-object, it is conceivable to embody them into the evaluation process, companion of the apply. This approach appears more amenable to efficient implementation (see the above comments on apply methods), since it draws on more traditional implementation technologies. However, it would blur evaluation, application and rewriting into one complex language processor where some interesting characteristics of the rewriting system would be lost.

The PRS is obviously a faithful, although incomplete, representation of the current program configuration. In fact, the PRS is a *declarative meta-level* for the introspection part of the object world, in the sense that it can be given a formal semantics in terms of an equational theory without dealing with the operational semantics of the reflective protocol. Declarative reflection is an area where little has been done, and our model appears to be a good testbed for further research on this subject. The second implementation alternative, which builds the whole language around priority rewriting, is appealing, especially to provide a formal account of the full language in terms of an equational theory. However, it raises several practical issues such as efficiency.

The third alternative could provide the best of both worlds. Both evaluation and the priority rewriting system for introspection would be kept independent of each other. We currently have an interpreter for our language implemented along this line. The major shortcoming of this approach is the cost of alternating between the core language processor and the rewriting system, since it happens at each method call. A good coordination between the object world and the PRS would be essential to the system, but this coordination is essentially what is called *causal connection* in reflection. The immediate consequence of preserving the rewrite system is that we could make it a first-class entity. This would have several concrete outcomes in terms of efficiency. First, causal connection would avoid reconstruction of the whole PRS each time a message is executed or each time an object is created or destroyed by simply updating the existing PRS. Second, rewriting rules could be used to implement optimizations. A cache for lookup methods could be viewed as a set of rewrite rules. In the Example 3, optimizing rewrite rules would have the form: $eval(mo, \mu, [], c(\gamma, [\gamma, mo], K)) \rightarrow eval(K, \delta, [lmo], ik)$. Such rules are simply shortcuts that bypass a chain of rewriting steps of a finite length. Thanks to priority rewrite systems, it would suffice to give them higher priority over conflicting rules to avoid ambiguities. The chief advantage of this approach would be to introduce optimizations without obscuring the implementation with details that can be expressed much more elegantly with rewrite rules.

## 5. Related Work

Behavioral reflection in object-oriented programming has been popularized by Maes in 3-KRS [17]. Watanabe and Yonezawa [27] have then studied the use of meta-objects in the concurrent reflective language ABCL/R, a work that initiated several others in that domain. Ferber [9] has suggested an integration of meta-objects into a class-based model, namely ObjVLisp [3].

Our approach shares with these the idea of implementing reflection in OOP by introducing a reflective introspection protocol on message passing. Maes and Ferber approaches to the potential infinite meta-regression are similar to ours; they impose explicitly in the language basic cases to fall back on the implementation language. Maes uses explicit calls to the implementation language through "Form" expressions that essentially quote Lisp expressions in 3-KRS. Ferber uses execution modes (reflect versus normal) and tests them explicitly to decide whether or not to reflect. Our approach proposes not to hard-code such tests into the language but to provide them as declarative knowledge in the priority rewrite system. We claim that this will prove to be easier to understand, and to modify.

Watanabe and Yonezawa [27], as most of the other work on actor-based reflection and also 3-Lisp, create meta-levels in a lazy fashion. In this approach, reflective computation is fired by an explicit call upon the meta-level, which is created on demand. If the meta-level is never accessed, it won't be created, hence stopping the meta-regression. This approach concurs with the original idea of reflection, which avoids to put an a priori limit on the *degree of introspection* [7] (a number of tower levels they need to be run), but it appears less amenable to efficient implementation. Work should be pursued in both directions in order to be able to fairly compare them after the emergence of a deeper understanding of behavioral reflection. We also believe that our work sheds some new light on these approaches.

Reflective towers are the cornerstone of behavioral reflection. 3-Lisp [7] makes them explicit, while 3-KRS [17] and CLOS [6] exhibits a similar characteristic. Our study has made explicit the link between object-oriented reflection and the functional one, first introduced by 3-Lisp but also studied in Scheme [26], [4]. The main differences between our approach and these are the following:

1. First, towers in our model appear in a method per method fashion and they are finite, giving them statically a fixed degree of introspection. This locality of effects ensures that only those methods that use a particular tower will be affected by the execution at the corresponding degree of introspection. This locality also suggests that optimizations may be easier.

2. Second, we don't make the single-threading assumption that enabled the work on meta-continuations [26], [4]. In those approaches, authors assume that only one level in the tower is executing at any time and then concentrate on level-shifting operations in potentially infinite towers. However, this prevents higher levels in the tower from making side-effects. Since we don't make this assumption, the control in our model cannot be captured by meta-continuations.

3. Third, reflective computations are not fired by climbing the tower through reflective functions but by making changes on apply methods themselves. We claim that this is dual to the 3-Lisp approach.

Notice also that the work on meta-continuations was driven by the goal of giving a formal account of reflection using denotational semantics. Even if it played an important role in the understanding of reflection, this approach finally failed because reflection impairs the compositionality assumption of denotational semantics [4]. Further work is still needed to achieve the goal of formalizing reflection and rewriting systems should play a role in such an attempt.

For instance, Mendhekar and Friedman [20] develop a programming logic based on the $\lambda_v$-calculus for a language in the line of 3-Lisp, Brown [26], and $\mathcal{I}_\mathcal{R}$ [13]. Reification is noted $\bigodot VU$, and the rewrite system that defines the language semantics reduces an applicative context $C[\bigodot VU]$ to $C[VV']$, where $V'$ is a representation of the current context $C$ whose redex is represented by $U$. Informally, $V$ can be viewed as a reflective function that is passed a representation of the current execution context. Coarsely speaking, our scheme could be rephrased in a functional programming style by attaching an apply function $\mathcal{F}$ to each abstraction $\mathcal{A}$ in a term $(\mathcal{A};\mathcal{F})$. Neglecting issues such as the order of evaluation, the application $((\mathcal{A};\mathcal{F})\ x)$ would then be preceded by an unfolding operation where the apply function would be extracted from $(\mathcal{A};\mathcal{F})$ and applied to the representation of $\mathcal{A}$ and $x$, e.g. $(\mathcal{F}\ f_R(\mathcal{A})\ f_R(x))$ where $f_R$ is a suitable representation function [22], [20]. The apply function would itself be an abstraction, and to stop the unfolding, one could provide a basic apply function, as in our model. The unfolding process could be expressed as a PRS with one rule to do the unfolding and a higher priority one to stop the unfolding on the basic apply function. A complete comparison of both systems, as well as with the generic functions implementation of Queinnec and Cointe [23], should be pursued in the next future.

## 6. Conclusion and Future Work

In this paper, we have studied a new model for behavioral reflection based on meta-objects in a prototype-based programming language. A new method invocation protocol, using the standard equation *message execution* = *lookup* $\circ$ *apply* augmented with first-class continuations, is the cornerstone of this model. We have given a formal semantics to this protocol using the theory of priority rewrite systems. We have then discussed the behavior of objects in the language by examining step by step the execution of a message. Not only does this study give a fine-grained understanding of behavioral reflection in our model, it also leads to two important conclusions concerning behavioral reflection in OOP.

First, our study confirms that the reification of the lookup does not cause fundamental problems and we can reasonably expect to implement it efficiently. Since it has many interesting applications in practice [10], this should be part of most object-oriented languages. Second, the reification of the apply methods leads to

reflective towers *à la* 3-Lisp, which confirms again the central role they play in behavioral reflection. Object orientation changes the perspectives by making towers local to each method and our model makes them finite. Nevertheless, this result suggests that no gain in efficiency can be obtained without deepening our understanding of reflective towers. Such gains will necessitate either a high-level and portable model for "compiling" apply methods or, if apply methods are definitional interpreters, the application of semantics-based program transformations techniques to be adapted to reflective towers.

While priority rewriting systems proved to be a successful approach to give a formal account of reflection in OOP, they can also provide a declarative meta-level to be reified and causally connected with the base language, leading to a new dimension of reflection. Priority rewrite systems are also highly successful in providing both a better understanding of reflection and a new framework for the efficient implementation of object-oriented reflective language.

This work points at several research directions. We still need to build libraries of examples using the reflective facilities discussed here, as well as developing a methodology for reflective software construction. An interpreter for our language is now available but efficient implementations are, in our view, one of the main research direction in the near future to make behavioral reflection an effective tool. Much work yet remains to obtain an implementation that would cope with this ambitious requirement, yet it is crucial for the future of reflective languages. Other directions of interest concern the use of priority rewrite systems. The research agenda includes work to be done to give a complete formal account of reflection where PRS could play a role. Including a rewriting meta-level and reflecting over it promises to be fruitful, both to include optimization rules and to be able to reflect on the reflective protocol itself.

## Acknowledgments

The authors wish to thank Jean Vaucher as well as anonymous referees for helpful comments on previous versions of this paper. We would also like to thank Marco Jacques who implemented the first interpreter for this language. Finally, the first author wishes to thank FCAR-Québec and NSERC-Canada for their support.

## Appendix

**Proposition 4** *For any program configuration $\mathcal{C}$ which respects the principles* `P1` *to* `P12`*, the PRS $\Delta(\mathcal{C})$ is* **terminating.**

**Proof:** The proof proceeds by induction on the number of objects in the program configuration $\mathcal{C}$.

**Basic case:** Consider the program configuration $\mathcal{C}$ corresponding to the kernel and any term $t \in \mathcal{T}(\mathcal{F}, \mathcal{H})$ to be l-reduced. Only ground terms of the form

$eval(o, s, a, k)$ and $blf(s, o)$ will be I-reduced. Terms of the form $blf(s, o)$ are I-reduced either to a term $m$ representing a method if $o$ represents an object in $\mathcal{C}$, or to the term $doesNotUnderstand$, and in both cases no further I-reduction is possible. Terms of the form $eval(o, s, a, k)$ can match many rules in $\Delta(\mathcal{C})$, but in the most general case, they are I-reduced using rule (14) (Table 2) to $eval(o, \mu, [], c(\gamma, [s, o], c(\alpha, [o, a, k], IK)))$. Since $o$ is an object in the kernel, by definition its meta-object is $BMO$ and by Def.7.1 $\Delta(\mathcal{C})$ contains a rule that I-reduces this term to $eval(c(\gamma, [s, o], c(\alpha, [o, a, k], IK)), \delta, [BMO], IK)$. Applying successively rules (14,1,4,6,5,14,2,4,7,5,10) as in Table 3 yields $eval(c(\alpha, [o, a, k], IK), \delta, [blf(s, o)], IK)$. By Def.7.2, $\Delta(\mathcal{C})$ contains a rule of the form $blf(s, o) \rightarrow m$ where $m$ represents a method in the kernel, which I-reduces this term to $eval(c(\alpha, [o, a, k], IK), \delta, [m], IK)$, then I-reduced to $eval(m, \alpha, [o, a, k], IK)$ by rule (5). Rule (14) applies and yields $eval(m, \mu, [], c(\gamma, [\alpha, m], c(\alpha, [m, [o, a, k], IK], IK)))$ Because $m$ represents a kernel method, by Def.7.1, $\Delta(\mathcal{C})$ contains a rule $eval(m, \mu, [], K) \rightarrow eval(K, \delta, [BMO], IK)$ which I-reduces this term to $eval(c(\gamma, [\alpha, m], c(\alpha, [m, [o, a, k], IK], IK)), \delta, [BMO], IK)$ This term is I-reduced to $eval(c(\alpha, [m, [o, a, k], IK], IK), \delta, [blf(\alpha, m)], IK)$ by rules (4,14,1,4,6,5,14,2,4,7,5,10) as in Table 3. Because $m$ is a method in the kernel, its apply method is $BA$ and by Def.7.2, $\Delta(\mathcal{C})$ contains a rule $blf(\alpha, m) \rightarrow BA$ that I-reduces the term to $eval(c(\alpha, [m, [o, a, k], IK], IK), \delta, [BA], IK)$. This last one I-reduces to $baf(m, o, a, k)$ by rules (5) and (9), and no further I-reduction is possible. This prove the proposition for the basic case.

**Induction step:** Assume now that the proposition is true for a program configuration $\mathcal{C}_n$ containing n objects. Consider a configuration $\mathcal{C}_{n+1}$ adding a $n+1$th object $o_{n+1}$ to $\mathcal{C}_n$. $\Delta(C_{n+1})$ is the same as $\Delta(C_n)$ (up to an isomorphism on $\overline{\mathcal{F}}$ and $\varphi$), except that there will be a new rule of the form $eval(o_{n+1}, \mu, [], K) \rightarrow eval(K, \delta, [mo_{n+1}], IK)$ and possibly new rules of the form $blf(s, o_{n+1})$ for all slots with name $s$ in $o_{n+1}$ if $mo_{n+1} = BMO$. If a message with selector $s'$, arguments $a'$ and current continuation $k'$ is sent to $o_{n+1}$, then rule (14) applies and I-reduces the term $eval(o_{n+1}, s', a', k')$ to $eval(o_{n+1}, \mu, [], c(\gamma, [s', o_{n+1}], c(\alpha, [o_{n+1}, a', k'], IK)))$. If $o_{n+1}$ does not correspond to an existing object, rule (13) will immediately I-reduce this term to its reduct $noSuchObject$. Otherwise, it is I-reduced by the meta-object fetching rule to $eval(c(\gamma, [s', o_{n+1}], c(\alpha, [o_{n+1}, a', k'], IK)), \delta, [mo_{n+1}], IK)$. By the induction hypothesis, the meta-object $mo_{n+1}$, its lookup method as well as the method $m'$ used to respond to the message already exist, so sending them messages yields terms that can be reduced in a finite number of I-reductions. ■

**Proposition 5** *For any program configuration $\mathcal{C}$ which respects the principles* P1 *to* P12*, the PRS $\Delta(\mathcal{C})$ is* **ground confluent.**

**Proof:** The proof proceeds by induction on the number of objects in the program configuration $\mathcal{C}$.

**Basic case:** $\Delta$ applied to the kernel produces a PRS whose rewrite rules are rules (4) to (14) in Table 2 augmented with meta-object fetching rules and rules whose left-hand sides (lhs) are of the form $blf(s, o)$. It is definitional, since the only defined

functions correspond to functors *eval* and $blf$ and, in the lhs of all rewrite rules, those functors never appear within arguments. The PRS is thus ground confluent by Proposition 1 because (1) there is no pair of rewrite rules with labels $h$ whose lhs unify, (2) all pairs of rewrite rules whose lhs unify contain one rule of higher priority than the others:

- All rules with label $h$ have their lhs unifying with either the lhs of rule (12) or the lhs of rule (14), but these two rules have lower priority.
- Rules (9) and (10) have their lhs unify, but (10) has higher priority.
- Rules (9) and (10) also have their lhs unify with the lhs of rule (14), but again rule (14) has lower priority.
- The lhs of rule (13) unifies with the meta-object fetching rules and with rule (14), but in each case conflicting rules don't have the same priority.

**Induction:** Assume the proposition true for a program configuration $\mathcal{C}_n$ with n objects. If we add an n+1th object $o$ to $\mathcal{C}_n$ to obtain a new configuration $\mathcal{C}_{n+1}$, $\Delta(\mathcal{C}_{n+1})$ is the same as $\Delta(\mathcal{C}_n)$ (up to an isomorphism on $\overline{\mathcal{F}}$, and $\varphi$), except that it adds one meta-object fetching rule if $o \in \mathcal{O}$ and rewrite rules whose lhs are of the form $blf(s, o)$ if the metaobject of $o$ is $BMO$. These new rules don't violate the conditions for $\Delta(\mathcal{C}_{n+1})$ to be definitional. They have priority $h$ and their lhs can only unify with the lhs of rules (12,13,14) having lower priority. ■

**Proposition 6** *For any program configuration $\mathcal{C}$ which respects the principles* `P1` *to* `P12`*, in the PRS $\Delta(\mathcal{C})$, every ground non-constructor term is reducible.*

**Proof:** The only two defined functions of the PRS, *eval* and $blf$, have "catch-all" rules that ensure the reducibility of every non-constructor term. ■

## Notes

1. Throughout the text, we use a Scheme-like syntax for the code. Message sending expressions are noted `(<receiver> <selector> <arg1> ...<argn>)`, where the subexpressions `<receiver>`, `<arg1>`, `...<argn>` reduces to objects while `<selector>` reduces to a symbol.
2. There should also be an environment, but since here we don't want to deal with evaluation of the receiver and the arguments, the environment doesn't need to be represented at this point.
3. A standard representation of list as first-order term uses the functor *cons* which mimics a cons with two arguments: the car and the cdr of the list. The 0-arity functor *nil* denotes an empty list. To lighten the text, we use the Prolog syntax for lists, i.e. a list $cons(1, cons(2, cons(3, nil)))$ is written out as $[1, 2, 3]$.

## References

1. A.W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. J.M.C. Baeten, J.A. Bergstra, and J.W. Klop. Term Rewriting Systems with Priorities. In *Rewriting Techniques and Appl.*, number 256 in LNCS, pages 83–94. Springer-Verlag, 1987.

3. P. Cointe. Metaclasses are First Class: the ObjVLisp Model. *Proc. of OOPSLA'87, ACM Sigplan Notices*, 22(12):156–167, December 1987.
4. O. Danvy and K. Malmkjaer. Intensions and Extensions in a Reflective Tower. In *Proc. of the 1988 ACM Symp. on Lisp and Functional Prog.*, pages 327–341, 1988.
5. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers, 1990.
6. J. des Rivières. The Secret Tower of CLOS. In *Informal Proc. of the First Workshop on Reflection and Metalevel Architectures in OOP, OOPSLA/ECOOP'90*, October 1990.
7. J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *Proc. of the 1984 ACM Symp. on Lisp and Functional Prog.*, pages 331–347, August 1984.
8. C. Dony, J. Malenfant, and P. Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. *Proc. of OOPSLA'92, ACM Sigplan Notices*, 27(10):201–217, October 1992.
9. J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. *Proc. of OOPSLA'89, ACM Sigplan Notices*, 24(10):317–326, October 1989.
10. B. Foote and R. E. Johnson. Reflective Facilities in Smalltalk-80. *Proc. of OOPSLA'89, ACM Sigplan Notices*, 24(10):327–335, October 1989.
11. U. Hölzle, B.-W. Chang, C. Chambers, and D. Ungar. The Self Manual, version 1.0. distributed with the Self software release, from Stanford University, July 1990.
12. IEEE, New-York. *IEEE Standard for the Scheme Programming Language*, IEEE Std 1178-1990 edition, 1991.
13. S. Jefferson and D.P. Friedman. A Simple Reflective Interpreter. pp. ??–??, this issue.
14. G. Kiczales. Making Reflection Safe for Real-World Users. In *Informal Proc. of the First Workshop on Reflection and Metalevel Architectures in OOP, OOPSLA/ECOOP'90*, October 1990.
15. G. Kiczales, J. Des Rivières, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
16. J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An Architecture for an Open Compiler. In [28], pages 95–106.
17. P. Maes. Concepts and Experiments in Computational Reflection. *Proc. of OOPSLA'87, ACM Sigplan Notices*, 22(12):147–155, December 1987.
18. J. Malenfant, P. Cointe, and C. Dony. Reflection in Prototype-Based Object-Oriented Programming Languages. In *Informal Proc. of the Second Workshop on Reflection and Metalevel Architectures in OOP, OOPSLA'91*, October 1991.
19. J. Malenfant, C. Dony, and P. Cointe. Behavioral Reflection in a Prototype-Based Language. In [28], pages 143–153.
20. A. Mendhekar and D.P. Friedman. Towards a Theory of Reflective Programming Languages. In *Informal Proc. of the Third Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'93*, October 1993.
21. C.K. Mohan. Priority Rewriting: Semantics, Confluence, and Conditionals. In *Rewriting Techniques and Applications*, number 355 in LNCS, pages 278–291. Springer-Verlag, 1989.
22. R. Muller. M-LISP: A Representation-Independent Dialect of LISP with Reduction Semantics. *ACM TOPLAS*, 14(4):589–615, October 1992.
23. C. Queinnec and P. Cointe. An Open-Ended Data Representation Model for Eu_Lisp. In *Proc. of the 1988 ACM Symp. on Lisp and Functional Prog.*, pages 298–308. June 1988.
24. B.C. Smith. Reflection and Semantics in Lisp. In *Proc. of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1984.
25. D. Ungar and R. Smith. Self: The Power of Simplicity. *Proc. of OOPSLA'87, ACM Sigplan Notices*, 22(12):227–242, December 1987.
26. M. Wand and D. P. Friedman. The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower. *Lisp and Symbolic Computation*, 1(1):11–37, 1988.
27. T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. *Proc. of OOPSLA'88, ACM Sigplan Notices*, 23(11):306–315, November 1988.
28. A. Yonezawa and B. Smith, editors. *Proc. of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architecture*. RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992.