Behavioral Reflection in a Prototype-Based Language

J. Malenfant, C. Dony and P. Cointe*

Département d'informatique et r.o., Université de Montréal, Montréal, Québec, CANADA

LIRMM, Université de Montpellier II, Montpellier, FRANCE

École des Mines de Nantes, Nantes, FRANCE

Paper category : Research.

Abstract

In [MCD92], we have contributed a reflective model for a prototype-based language based on the *lookup* ∘ *apply* reflective introspection protocol. Here, we pursue this work by including a reification of continuations. Hence, we provide continuations as first-class objects and convert our previous protocol to handle them. First-class continuations provide much more control over the current computation. Also, this new model establishes the clear link between reflection in object-oriented and reflective towers as examplified by 3-Lisp [Smi84]. Object-orientedness provides reflection a more principled and encapsulated programming style, making it easier to use. In this paper, we establish the correctness of this new model, namely that any message in the system will be executed in a finite number of computation steps. This fact is based on a limited number of hypothesis independent from the implementation but from which we derive fundamental clues to derive an evaluator for the language.

1 Introduction

Reflection, understood as the construction of self-awared systems, is a persistent source of challenge. The mere feeling of touching the essence of computing, but also the tremendous potential for new applications insure a continuous quest for understanding its foundations. The goal of this paper is to propose and study in depth a minimal model of behavioral reflection for a prototype-based language.

Object-oriented programming is dominated by the class and metaclass approach that provides it a highly satisfactory solution to the problem of *structural reflection*, which is concerned by the complete reification of data structures and programs as first class entities. The problem of *behavioral reflection*, concerned by the reification of objects' execution, is not established on similar firm grounds yet.

*Authors current addresses : Département d'informatique, Université de Montréal, C.P. 6128, Succursale A, Montréal, Québec, Canada H3C 3J7, phone : (514) 343-7479, e-mail : malenfan@iro.umontreal.ca — LIRMM, 860 Rte de Saint-Priest, 34090 Montpellier, France, e-mail: dony@lirmm.fr — École des Mines de Nantes, 3 rue Marcel Sembat, 44049 Nantes Cedex 04, France, e-mail : pc@litp.ibp.fr

To solve the problem of behavioral reflection several approaches are currently investigated. Meta-objects, for instance, have been proposed in the frame language 3-KRS [Mae87] and then in an actor language [WY88] as well as in a class-based language [Fer89], to represent the behavioral properties of objects. Several problems are actually 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)?

In this paper, we mainly address the three first issues. Our goal is to concentrate on meta-object based behavioral reflection and to push this idea to its limit. We also use a prototype-based language as testbed in order to avoid the unnecessary complexity of classes before we fully understand behavioral reflection. In fact, 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 [DMC92, MCD92].
- Structural reflection is also provided in a minimal way, just to make behavioral reflection work properly.
- We choose to cope with the potential infinite regression of meta-objects by providing a basic meta-object which admits itself as its own meta-object.

To overcome some limitations of our previous protocol [MCD92], we also consider the reification of continuations as first-class objects in our prototype-based language. Continuations give much more control over the current computation. By converting our protocol to handle first-class continuations, we make apply methods capable to examine, modify or otherwise deal with them at run-time. This new *lookup* ∘ *apply* reflective protocol augmented with a reification of continuations matches the object-oriented computational model very well and provides better reflective capabilities. When modifying the lookup for one object or the application of a method, we now have control over the rest of the computation as in 3-Lisp [Smi84] or in Scheme [IEE91] and its behaviorally reflective extensions [JF92].

In this paper, our main goal is to fully investigate this new approach to assess its feasability and to highlight its

properties. Alternatives to this approach are investigated [Fer89, Coi90, WY88]. We believe that our work sheds some new light on these approaches.

The outline of the paper is the following. In the next section, we present the prototype-based language we are starting from and we add it sufficient structural reflection capabilities to study behavioral reflection. In Section 3, we describe our behavioral reflection model while in Section 4, we define it more systematically to study its feasability and its properties. In Section 5, we discuss the general object behavior and the implementation of the resulting reflective prototype-based programming language. We then conclude and discuss future work.

2 Reflective Prototypes

In this section, we describe a minimal prototype-based language that we extend with structural reflection capabilities.

2.1 A Minimal Language

In [DMC92], 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). Slots can be either private or public ; a private slot of an prototype `P` can only be accessed within `P` or in one of its extensions (see below).

P2 : Message passing is the only mean to activate a prototype and slots names are used as selectors in messages. No difference is made between data slots and method slots, both are accessed through messages.

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 newInitials: initform` 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*). Again, we refer readers to [DMC92] for more details. Finally, we assume that prototypes have only one parent ; this restriction could be relaxed, but multiple parents adds nothing to our study except an unnecessary complexity.

2.2 Structural Reflection

How can we provide structural reflection in this basic language? In class-based languages, classes and metaclasses play the role of structural reflection but in prototype-based languages, there is no more classes to deal with the representation of objects ; an alternative must be sought to obtain similar capabilities.

In fact, prototypes are not easily amenable to structural reflection [MCD91]. The simple fact 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.

Since studying behavioral reflection only need 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 [MCD91, HCCU90] five primitive access functions : *size(o)* to get the size of an object *o*, *name(o,i)* to get the name of its *ith* individual slot, *contentsAt(o,i)* to get the value of its *ith* slots, *contentsAtPut(o,i,v)* to set the value of its *ith* slot, and *isMethodAt(o,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 : `Size`, `NameAt`, `ContentsAt`, `ContentsAtPut`, and `IsMethodAt`. The object `ROOT` gets these methods, to which it points through its method slots : `size`, `nameAt:`, `contentsAt:`, `contentsAt:put:`, and `isMethodAt:` 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.

3 Behavioral Reflection

In this section, we first describe how the behavior of objects is described, how the user can modify this behavior, and what are the basic objects implementing the standard behavior.

3.1 Method Invocation Protocol

There are two main aspects in the behavioral reflection model. First, it must describe the behavior of objects using other objects. Second, it must provide a method invocation protocol that will allow the user to intervene on the current execution in order to modify the course of events, i.e. to reflect.

To describe the behavior of objects, we associate a metaobject to each of them, whose function is to explicit how the object reacts when it receives a message. Since the object-oriented model of computation uses message passing as its fundamental mechanism, it is usual to make message sends the vantage points 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 :

$$message\ execution = lookup \circ apply$$

Hence, these operations can be implemented as methods, themselves reified as objects in the language, allowing the user to redefine them in order to perform reflective computations. At first look, each message passing operation `o sel: a1 ect: a2 or: a3` is replaced by a reflective introspection which has three phases : (1) find the meta-object of the receiver `o`, (2) send it a message `lookup: (#sel:ect:or:) in: o`[1] that yields a method object (3) to which is sent a message `applyTo: o withArgs: #(a1 a2 a3) withCont: k`. Hence, coarsely speaking, the reflective introspection rule is :

```
o sel: a1 ect: a2 or: a3 ⇒
(((o metaObject)
   lookup:(#sel:ect:or:)  in: o)
   applyTo: o withArgs: #(a1 a2 a3) withCont: k)
```

Obviously, each of these operations can lead to potential infinite meta-regression. Since all three are represented as message sends, the same introspection rule can be applied to each of them *ad infinitum*. The evaluator will be responsible of preventing this to happen, as we will see (§4), and basic cases will be provided to represent its standard behavior.

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 that is able to answer an apply message in order to make the reflective introspection rule working properly. Since 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. Nevertheless, we have decided to force the lookup phase to return an object when a message is accessing a data slot but it must be created on the fly, in a lazy fashion. They answer apply messages by simply returning the value of their corresponding data slot. Notice that this property must always hold in the system and, as we will see (§3.2), the kernel of the language will insure it through its primitive lookup function.

First-class continuations

First-class continuations now represent a long tradition, especially in the Lisp and Scheme community [Ste90, IEE91]. In our context, continuations are objects, or prototypes, representing the (default) future of the computation at a given point in the execution of the program. Moreover, most of the time, the current computation step will return an object as its result, and the first thing to do after that is to send this object a message. Thus, although continuations may be represented in many different ways, here we simply assume that they are objects applied by sending them a message `applyContTo:withArgs:` :

```
k applyContTo: o withArgs: #(a1 a2 ...aj)
```

In the standard case, continuations will have only one argument, the object `o`, and the effect of this message will be to send `o` the message representing the next thing to do in the computation. In this case, the array of arguments will be empty. We maintain the form `applyContTo:withArgs:` to keep its full generality to our approach.

[1] Readers familiar with Smalltalk-80 should recognize its syntax where the special character # is used both to quote symbols and arrays.

3.2 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 answer to lookup messages ; meta-objects can be shared among several objects and behave like local interpreters for them.

P8 : Meta-objects answer lookup messages by returning method objects that are able to answer 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, (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 proposes to solve the problem 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 [Coi87].

Because of the method invocation protocol, `BasicMetaObject`, as any other meta-object in the system, must be able to answer `lookup:in:` message. In fact, since it gives the standard behavior, its lookup method is the primitive lookup function reified as a method in the language. We call this method object `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. We call this method object `BasicApply` and construct it to have itself as its own apply method. In the same way, reifying continuation requires the introduction of a primitive function to apply them, reified in the language. We call this method object `BasicApplyCont`, and construct it to have `BasicApply` as apply method.

These solutions add again four principles to the reflective prototype-based language :

P9 : `BasicMetaObject` is the first meta-object in the system ; it admits itself as its own meta-object.

P10 : `BasicLookup` is the first lookup method in the system ; it represents the primitive lookup function reified as a method object in the language.

P11 : `BasicApply` is the first apply method in the system ; it represents the primitive apply function reified as a method object in the language.

P12 : `BasicApplyCont` is the first method in the system to apply continuations ; it represents the primitive function to apply continuations, reified as a method object in the language.

The kernel of our model is constructed around six basic objects : `BasicMetaObject`, `BasicLookup`, `BasicApply`, `BasicApplyCont`, `IK` and `ROOT` (see §2). `IK` is an object representing the identity continuation whose behavior when applied is to return its first argument as result ; it also provides

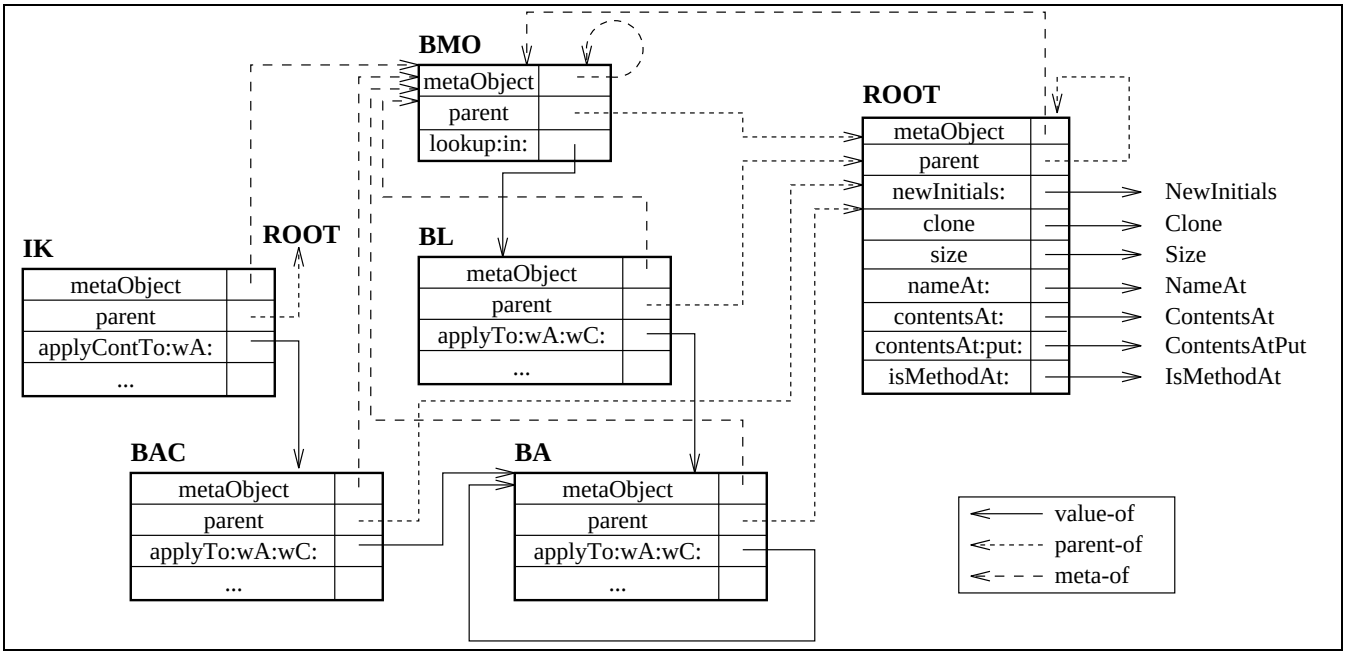

Figure 1: Basic objects and their relationships.

the basic methods for continuations. The Figure 1 illustrates the kernel objects as well as their relationships.

4 Correctness

In this section, we define our model using some shorthand notation that we use to establish its correctness and to become acquainted with its overall behavior.

4.1 The Model and its Evaluator

Our goal in this section is to show that the fundamental characteristics of the model and the necessary conditions to write a correct evaluator for our reflective prototype-based programming language can be captured by a limited number of hypothesis. We don't try to give a complete formal proof of this, but only sufficient insights to convince that the model is correct. From these, we will gain information for the implementation of the language.

4.1.1 Model Hypothesis

In our language, everything is represented using objects, or prototypes. A running program consists in a set of objects. Some of them are simply end-user objects but others are used by the evaluator to actually run the program : method objects, continuation objects, selectors reified as symbol objects, object vectors used to pass arguments to methods, and naturally meta-objects that can be created by users to modify the behavior of end-user objects.

As we have seen above, the method invocation protocol performs three main operations which are designated by three message selectors : `metaObject` to find the meta-object of the receiver, `lookup:in:` to find the method corresponding to the selector of the message, and `applyTo:withArgs: withCont:` to apply this method to the receiver with the arguments of the message. In the rest of the section, we use the following shorthand notation for these fundamental operations :

`o sel: a1 ect: a2 or: a3` $\rightarrow o.s(a_1, a_2, a_3)$ *where* $s =$ `sel:ect:or:`

`o metaObject` $\rightarrow \mu(o)$

`mo lookup: s in: o` $\rightarrow mo.\gamma(s, o)$

`m applyTo: o withArgs: #(a1,...,aj) withCont: k` $\rightarrow m.\alpha(o, [a_1, \ldots, a_j], k)$

`k applyContTo: o withArgs: #(a1,...,aj)` $\rightarrow k.\delta(o, [a_1, \ldots, a_j])$

The message sending operator '.' is left-associative. The primary goal of this notation is to keep proofs small enough to be understandable. Using the normal syntax would rapidly lead to useless fullpage expressions.

Provided with these basic definitions and notation, we now switch on the characteristics of the model. These characteristics can be expressed in eight statements that we will call "model hypothesis". The first model hypothesis states that for each object, there is an associated meta-object :

HM1 *Each object has a meta-object, i.e.*

$$mx = \mu(x)$$

where mx is called the meta-object of x.

The meta-object of an object is used in the reflective introspection rule that is stated as follows :

HM2 *Reflective introspection. The reflective introspection rule is :*

$$o.s(a_1, \ldots, a_j) \Rightarrow \mu(o).\gamma(s, o).\alpha(o, [a_1, \ldots, a_j], k)$$

where k represents the continuation of the computation appearing after the message send. This rule makes no assumptions about the mechanism used to evaluate the receiver o, the selector s and the arguments $a_1, \ldots, a_j$ *except that they are reified into objects.*

In fact, we hide things a bit here since the lookup and apply functions are only two subparts of the complete evaluation process. When applying the reflective introspection rule, the receiver, the selector and the arguments of the message may be evaluated. Because we do not address this problem, we only assume that their representation is reified into 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.

The model provides a basic meta-object constructed to be its own meta-object :

HM3 *The existence of a basic meta-object. There exists one object such that :*

$$BMO = \mu(BMO)$$

This object is called BasicMetaObject, or BMO for short.

Since all meta-objects must answer `lookup:in:` messages, `BasicMetaObject` gets its own lookup method. This method is a primitive lookup function, reified as a standard method in the language :

HM4 *The existence of a basic lookup. There exists one method object such that :*

$$BL = BMO.\gamma(\gamma, BMO)$$

This method is called BasicLookup, or BL for short. Furthermore, the object BL is constructed to get $\mu(BL) = BMO$.

Since `BasicLookup` is a method object, it gets its own apply method. Again, this method is a primitive apply function, reified as a standard method in the language :

HM5 *The existence of a basic apply. There exists one method object such that :*

$$BA = BMO.\gamma(\alpha, BL)$$

This method is called BasicApply, or BA for short. Furthermore, the object BA is constructed to get $\mu(BA) = BMO$ and $BMO.\gamma(\alpha, BA) = BA$.

Objects are related through parent-of links and there is one root object that holds as its own methods the primitive of the language :

HM6 *The existence of a root object. There exists one object such that :*

$$ROOT = \pi(ROOT)$$

The object ROOT is constructed to get $\mu(ROOT) = BMO$ and such that its methods represent the primitives of the language, themselves reified as method objects and constructed in such a way that BMO is their meta-object and BA their apply method.

The primitive lookup function takes advantage of the fact that `ROOT` is its own parent to stop looking for a parent when encountering it. We define `ROOT` like this because we assume that any object in the system can anwer a message `parent` without raising an error. The usual alternative to this is to make `nil` the parent of the root object, but since we are committed to being reflective, this would force to represent `nil` as an object and to go back to the similar problem, namely what is the parent of `nil`?

Finally, the kernel provides a first continuation object, the identity continuation exhibiting the primitive behavior of continuations :

HM7 *The existence of an identity continuation. There exists one object such that :*

$$o = IK.\delta(o, [])$$

This continuation object represents the simplest continuation which merely returns the object to which it is applied. IK is used by the reflective introspection rule if nothing has to be done after the current message send (see HM2).

Since continuations must be applied, the kernel also provides a basic apply method for continuations :

HM8 *The existence of a basic apply for continuations. There exists method one object such that :*

$$BAC = BMO.\gamma(\delta, IK)$$

This method object is called BasicApplyCont, or BAC for short. Furthermore, the object BAC is constructed to get $\mu(BAC) = BMO$ and $BMO.\gamma(\alpha, BAC) = BA$.

4.1.2 Evaluator Hypothesis

At this point, the above hypothesis are sufficient to describe the kernel objects of Figure 1. However, they are insufficient to build a correct evaluator for the language. In order to get an operationally meaningful language, we need four more hypothesis. Since they have nothing to do with the model itself but rather with its operational semantics, we call them "evaluator hypothesis". Each of them cope with one particular problem with the execution of the model.

The first problem that comes up is the meta-regression in accessing an object's meta-object. Since now, we have assumed the object to have a slot called `metaObject` pointing to its meta-object. A natural way to access this meta-object is to send a message `metaObject` to the object, as suggested by the reflective introspection rule.

We must notice immediately that since this is a message, the method invocation protocol applies and ask again for the meta-object of the receiver. Hence, we are in a typical meta-regression. To get out of this, we make the access to the meta-object a primitive operation instead of a normal message :

HE1 *Primitivity of μ. The meta-object accessing rule is :*

$$\mu(o) \Rightarrow metaof(o) = mo$$

where mo is the meta-object of o. Hence, finding the meta-object of an object is a primitive function of the system called metaof. For the sake of brievity, we will use the following notation:

$$\underbrace{\mu(\mu(\ldots\mu}_{n\ times}(o)\ldots)) = m^n o$$

The second problem is how to connect the method object `BasicApply` and the primitive apply function *ba*. In the kernel, `BasicApply` gets itself as its own apply method. To be operational, the evaluator must use this property to find out when the function *ba* must be applied. The second evaluator hypothesis copes with this :

HE2 *The BasicApply apply rule is :*

$$BA.\alpha(m, [o, args, k_0], k_1) \Rightarrow k_1.\delta(ba(m, o, args, k_0), [])$$

Applying the apply method of BA to BA with the arguments m, o, args, k_0 and k_1 is equivalent to applying the evaluator primitive apply function ba to m, o, args and k_0 and apply the continuation k_1 to its result.

The next problem is how to relate the method object `BasicLookup` and the primitive lookup function *bl*. The following hypothesis can be seen as an implementation constraint that must hold to get a correct behavior :

HE3 *BasicLookup applying. The lookup method BL and the primitive lookup function bl are equivalent :*

$$ba(BL, BMO, [s, o], k) \Rightarrow k.\delta(bl(s, o), [])$$

Applying the evaluator ba function to the method object BasicLookup in the context of BMO with arguments s, o and k is equivalent to applying the evaluator primitive lookup function bl to s and o and applying k to its result.

Applying continuations is the last problem where a potential infinite regression must be prevented. A continuation object `K` is applied by sending it a message `applyContTo: withArgs:`. If we blindly use the reflective introspection rule to execute this message, it will create more continuations to be applied, which will need again other continuations to be applied, and so on. To prevent such a situation, we assume that continuations created by the basic evaluator, such as those introduced by the reflective introspection rule, are directly recognizable and executable :

HE4 *Application of primitive continuations. A primitive continuation K, such as those created by the application of the reflective introspection rule (see HM2), is directly executable by the evaluator :*

$$K.\delta(o, []) = o.K$$

In fact, we simplify a bit again here. K is an object reifying a continuation, and continuations created by the evaluator when applying the reflective introspection rule are all of the same kind, i.e. they accept only one argument : an object to which a message is sent. For the sake of brievity, we will assimilate the continuation K and the message it sends to its object argument o.

4.2 Lemmas and Theorems

We now show that the above hypothesis are sufficient to get a correct evaluator for the language. A side goal of this section is to master how computation are actually made in this model.

4.2.1 Fundamental Lemmas

Correctness corresponds here to the ability of an evaluator to execute any message in the language obtained from the model using a finite number of steps, that is a finite number of applications of defined rules and a finite number of computational steps by the evaluator.

To establish that, we first consider the kernel objects and show that any message to one of them is executed in a finite number of steps. Indeed, we assume that the primitive functions *ba* and *bl* are executed in a finite number of computation steps. Hence, it suffices to establish that any message to one of the objects in the kernel involves only a finite number of steps.

We proceed in four lemmas, followed by a first theorem establishing the correctness of the kernel under its basic hypothesis. The first lemma shows that an apply message sent to the method object `BasicLookup` is rewritten in a finite number of steps into a call to the primitive lookup function :

Lemma 1 *Applying the BasicLookup method object is equivalent to executing the evaluator lookup function bl, i.e.*

$$BL.\alpha(BMO, [s, o], k) = k.\delta(bl(s, o), [])$$

Proof [2].

$$\begin{array}{rcl} & & BL.\alpha(BMO, [s, o], k) \\ (HM2) & \Rightarrow & \mu(BL).\gamma(\alpha, BL).\alpha(BL, [BMO, [s, o]k], IK) \\ (HM4) & = & BMO.\gamma(\alpha, BL).\alpha(BL, [BMO, [s, o], k], IK) \\ (HM5) & = & BA.\alpha(BL, [BMO, [s, o], k], IK) \\ (HM2) & = & IK.\delta(ba(BL, BMO, [s, o], k), []) \\ (HM7) & = & ba(BL, BMO, [s, o], k) \\ (HE3) & = & k.\delta(bl(s, o), []) \quad \square \end{array}$$

Second, we establish that an apply message sent to the method object `BasicApply` is rewritten in a finite number of steps into a call to the primitive apply function :

Lemma 2 *Applying the BasicApply method is equivalent to executing the evaluator apply function ba, i.e.*

$$BA.\alpha(m, [o, a, k], IK) = ba(m, o, a, k)$$

Proof.

$$\begin{array}{rcl} & & BA.\alpha(m, [o, a, k], IK) \\ (HE2) & = & IK.\delta(ba(m, o, a, k), []) \\ (HM7) & = & ba(m, o, a, k) \quad \square \end{array}$$

Given these two first lemmas, we know that `BasicLookup` an `BasicApply`, two of the six objects in the kernel behave correctly when they are sent their standard messages. Now, let's look at `BasicMetaObject`, which mainly answers `lookup:in:` messages. The following lemma shows that a lookup message sent to `BMO` is also rewritten in a call to the primitive lookup function in a finite number of steps :

Lemma 3 *Sending BasicMetaObject the lookup message is equivalent to applying the evaluator's primitive lookup function bl :*

$$BMO.\gamma(s, o) = bl(s, o)$$

Proof.

$$\begin{array}{rcl} & & BMO.\gamma(s, o) \\ (HM2) & \Rightarrow & \mu(BMO).\gamma(\gamma, BMO).\alpha(BMO, [s, o], IK) \\ (HM3) & = & BMO.\gamma(\gamma, BMO).\alpha(BMO, [s, o], IK) \\ (HM4) & = & BL.\alpha(BMO, [s, o], IK) \\ (L1) & = & IK.\delta(bl(s, o), []) \\ (HM7) & = & bl(s, o) \quad \square \end{array}$$

At this point, we have established that `BL`, `BA` and `BMO` behave correctly when they are sent the messages used by the method invocation protocol. Now, we turn to the other method objects in the kernel. All method objects in the kernel are *standard methods*. A standard method object `m` is characterized by the two following statements :

[2] Proofs are all based on a similar pattern and are quite simple. They consist in applying the right rule or one of the basic cases to derive the conclusion from the premises. Each step is labelled with either the hypothesis or the lemma used to transform the above expression into a new expression.

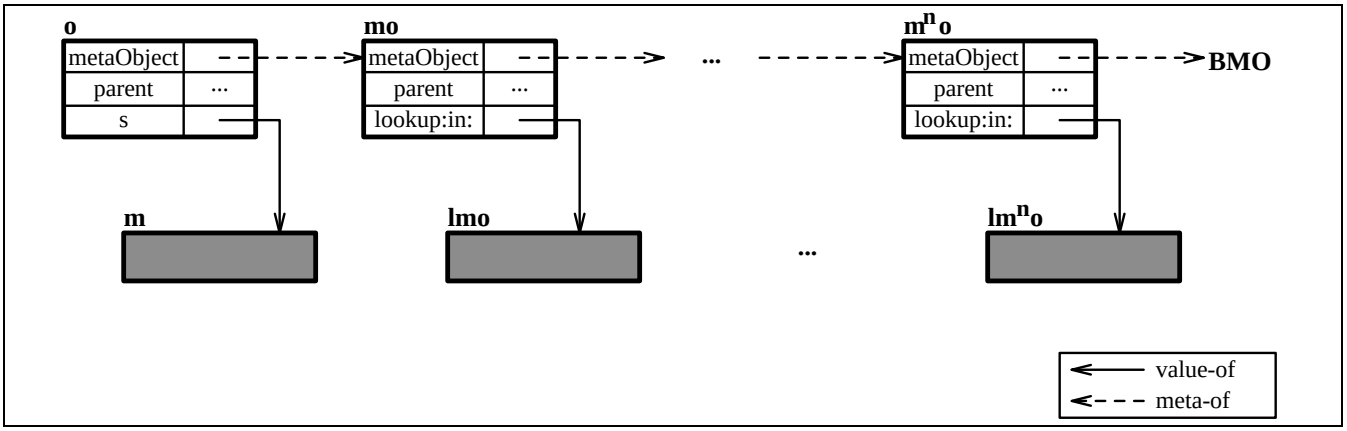

Figure 2: General object description.

S1. $\mu(m) = BMO$, i.e. its meta-object is the basic meta-object.
S2. $bl(\alpha, m) = BA$, i.e. its apply method is the basic apply method.

In the following lemma, we show that a standard method answers an apply message in a finite number of steps :

Lemma 4 *Sending an apply message to a standard method is equivalent to executing this method using the primitive apply function :*

$$m.\alpha(o, [a_1, \ldots, a_j], k) = ba(m, o, [a_1, \ldots, a_j], k)$$

where j is the number of arguments of m.

Proof.

$$\begin{array}{rcl} & & m.\alpha(o, [a_1, \ldots, a_j], k) \\ (HM2) & \Rightarrow & \mu(m).\gamma(\alpha, m).\alpha(m, [o, [a_1, \ldots, a_j], k], IK) \\ (S1) & = & BMO.\gamma(\alpha, m).\alpha(m, [o, [a_1, \ldots, a_j], k], IK) \\ (L3) & = & bl(\alpha, m).\alpha(m, [o, [a_1, \ldots, a_j], k], IK) \\ (S2) & = & BA.\alpha(m, [o, [a_1, \ldots, a_j], k], IK) \\ (L2) & = & ba(m, o, [a_1, \ldots, a_j], k) \quad \square \end{array}$$

Recall that messages that access data slots are also answered through the same method invocation protocol (see §3). We simply assume that, in this case, the primitive lookup function returns standard methods that themselves return the value of the corresponding slot when applied. Thus, accessing a data slot is also done in a finite number of steps. Indeed, this property must be maintained in any extension of the kernel to build applications.

Theorem 1 *Sending a message to any object in the kernel is equivalent to applying the primitive apply function ba :*

$$o.s(a_1, \ldots, a_j) = ba(m, o, [a_1, \ldots, a_j], k)$$

where m = bl(s,o), the method object corresponding to the selector s in o, and k represents the rest of the computation after $o.s(a_1, \ldots, a_j)$.

Proof.

$$\begin{array}{rcl} & & o.s(a_1, \ldots, a_j) \\ (HM2) & \Rightarrow & \mu(o).\gamma(s, o).\alpha(o, [a_1, \ldots, a_j], k) \\ & = & BMO.\gamma(s, o).\alpha(o, [a_1, \ldots, a_j], k) \\ (L3) & = & bl(s, o).\alpha(o, [a_1, \ldots, a_j], k) \\ & = & m.\alpha(o, [a_1, \ldots, a_j], k) \\ (L4) & = & ba(m, o, [a_1, \ldots, a_j], k) \quad \square \end{array}$$

The second step uses the fact that all objects in the kernel have BMO as meta-object while the fifth step uses the fact that all methods in the kernel are standard ones.

4.2.2 General Case

Being done with the kernel prototypes, we now attack the general case, when new objects are added to the kernel to build applications. The next theorem establishes the correctness of the resulting system. We assume that the protocol underlying the kernel is respected by the programmer and that the extensions are correct.

For example, we assume that user methods are well-behaved, e.g. they do not get into infinite loops. And by correct extension of the kernel, we mean that it follows the principles established by the model, e.g. every new object must have a meta-object capable of answering `lookup:in:` messages, and that method objects are able to answer `applyTo:withArgs:withCont:` messages.

Given that these assumptions are true, any message to any object in the system is executed in a finite number of steps.

Theorem 2 *A message sent to any object in a system extending correctly the kernel is executed in a finite number of steps.*

Proof. The proof proceeds by induction. The Theorem 1 establishes the basic case, i.e. the statement is true for kernel objects. Suppose it true for a system consisting of N objects, then if we add a (N+1)th object o, and send it a message $o.s(a_1, \ldots, a_j)$. Then, the reflective introspection rule apply, and we get $\mu(o).\gamma(s, o).\alpha(o, [a_1, \ldots, a_j], k)$. By the evaluator hypothesis HE1, we get $\mu(o)$ in one step. But the meta-object of o exists and by the induction hypothesis, the message $\gamma(s, o)$ yields the method object m, corresponding to the selector s in o, in a finite number of steps. Again, this method object exists and by the induction hypothesis, the message $\alpha(o, [a_1, \ldots, a_j], k)$ is also executed in a finite number of steps before proceeding with the continuation k. $\square$

$$
\begin{array}{rcll}
 & o.s(a_1,\ldots,a_j) & & (1)\\
(HM2) & \Rightarrow & \mu(o).\gamma(s,o).\alpha(o,[a_1,\ldots,a_j],k) & (2)\\
(HE1) & \Rightarrow & mo.\gamma(s,o).\alpha(o,[a_1,\ldots,a_j],k) & (3)\\
(HM2) & \Rightarrow & \mu(mo).\gamma(\gamma,mo).\alpha(mo,[s,o],\alpha(o,[a_1,\ldots,a_j],k)) & (4)\\
(HE1) & \Rightarrow & m^2o.\gamma(\gamma,mo).\alpha(mo,[s,o],\alpha(o,[a_1,\ldots,a_j],k)) & (5)\\
 & & \ldots & \\
(HM2) & \Rightarrow & \mu(m^no).\gamma(\gamma,m^no).\alpha(m^no,[\gamma,m^{n-1}o],\ldots\alpha(m^2o,[\gamma,mo],\alpha(mo,[s,o],\alpha(o,[a_1,\ldots,a_j],k)))\ldots) & (6)\\
(HP1) & = & BMO.\gamma(\gamma,m^no).\alpha(m^no,[\gamma,m^{n-1}o],\ldots\alpha(m^2o,[\gamma,mo],\alpha(mo,[s,o],\alpha(o,[a_1,\ldots,a_j],k)))\ldots) & (7)\\
(L3) & = & bl(\gamma,m^no).\alpha(m^no,[\gamma,m^{n-1}o],\alpha(m^{n-1}o,[\gamma,m^{n-2}o],\ldots\alpha(m^2o,[\gamma,mo], & (8)\\
 & & \alpha(mo,[s,o],\alpha(o,[a_1,\ldots,a_j],k)))\ldots)) & \\
(HP2) & = & lm^no.\alpha(m^no,[\gamma,m^{n-1}o],\alpha(m^{n-1}o,[\gamma,m^{n-2}o],\ldots\alpha(m^2o,[\gamma,mo],\alpha(mo,[s,o],\alpha(o,[a_1,\ldots,a_j],k)))\ldots)) & (9)\\
(HP3) & = & lm^{n-1}o.\alpha(m^{n-1}o,[\gamma,m^{n-2}o],\ldots\alpha(m^2o,[\gamma,mo],\alpha(mo,[s,o],\alpha(o,[a_1,\ldots,a_j],k)))\ldots) & (10)\\
 & & \ldots & \\
(HP3) & = & lmo.\alpha(mo,[s,o],\alpha(o,[a_1,\ldots,a_j],k)) & (11)\\
(HP4) & = & m.\alpha(o,[a_1,\ldots,a_j],k) & (12)
\end{array}
$$

Figure 3: General case for $o.s(a_1,\ldots,a_j)$ (lookup phase).

Hence, if the basic rules of this system and its behavioral reflection protocol are followed in building applications, we will get a correct system. Over this basic result, let's go back to the operational behavior of the system and look at how to implement it.

5 Implementation Concerns

Although it is important to establish the correctness of the model, this gives little hints about the general behavior of the system. An interesting outcome of the above exercise lies in its implications on the implementation of the language evaluator. We now have a look at these issues.

5.1 General object behavior

A major concern about meta-level and reflective architectures is efficiency. Hence, before looking at the implementation of the kernel, let's consider the behavior of our model when a message is sent to some general object, to identify the potential sources of inefficiency and to discuss their implementation.

First, generally speaking, an object has a meta-object, which itself has a meta-meta-object, and so on until `BMO` is reached. We assume there are n levels of meta-objects before `BMO`. As requested by the method invocation protocol, each of these meta-objects has a lookup method. The meta-object of the deepest meta-object m^no is `BMO`, thus the primitive lookup applies to it, and if we ask for its lookup method, we find a method object lm^no.

The other $n-1$ meta-objects also have their own lookup methods, called respectively lmo, lm^2o, ..., $lm^{n-1}o$ that we can find by applying to them the lookup method of their respective meta-objects.

For the purpose of this study, assume that we send the message `s` to `o` and that the correponding method object is `m`. The Figure 2 illustrates a possible configuration for such a general object `o`, its related meta-objects as well as their respective lookup methods.

To make the problem tractable, we describe these characteristics using again hypothesis that we call *problem hypothesis*. The first four hypothesis formalize the construction illustrated in Figure 2 :

HP1 *The object o has has a meta-object mo that in turn has a meta-object m^2o, and so on for n level, while the meta-object at the (n+1)th level is BMO, i.e.*

$$
\begin{array}{rcl}
\mu(o) & = & mo\\
\mu(m^io) & = & m^{i+1}o,\ 1\le i<n-1\\
\mu(m^no) & = & BMO
\end{array}
$$

HP2 *The lookup method of the nth level meta-object is called lm^no, i.e.*

$$bl(\gamma,m^no)=lm^no$$

HP3 *The lookup method lm^io associated to the ith level meta-object m^io is found when applying the lookup method $lm^{i+1}o$ associated to the (i+1)th level meta-object $m^{i+1}o$, i.e.*

$$lm^{i+1}o.\alpha(m^{i+1}o,[\gamma,m^io],k)=k.\delta(lm^io,[]),1\le i\le n-1$$

and if k is a continuation created by the application of HM2, then $k.\delta(lm^io,[])=lm^io.k$ by HE4.

HP4 *m is the method object found when applying the lookup method lmo to the selector s and the object o, i.e.*

$$lmo.\alpha(mo,[s,o],k)=k.\delta(m,[])$$

and if k is a continuation created by the application of HM2, then $k.\delta(lmo,[])=lmo.k$ by HE4.

These four hypothesis are sufficient to consider what happens in the lookup phase of the message execution. The Figure 3 develops the reflective equations for this first phase.

When the method object `m` has been found, we send it the message `applyTo:withArgs:withCont:`. Before addressing

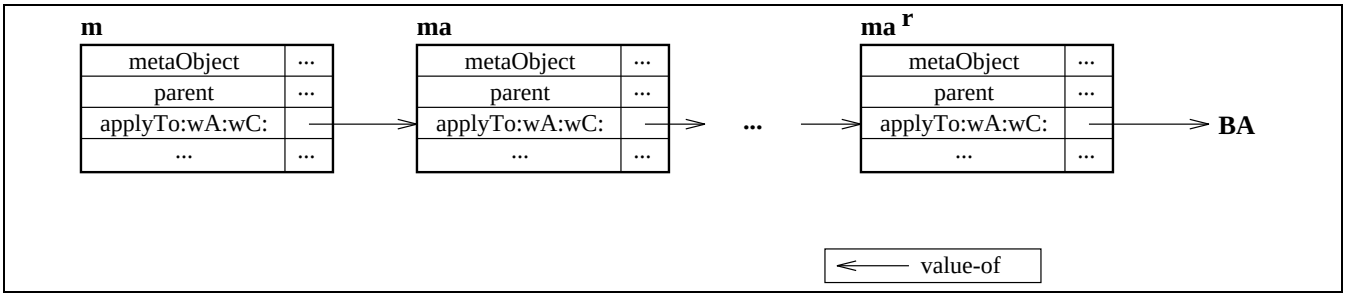

Figure 4: General method application description.

$$
\begin{aligned}
& m.\alpha(o,[a_1,\ldots,a_j],k) && (13)\\
(HM2) \quad \Rightarrow \quad & \mu(m).\gamma(\alpha,m).\alpha(m,[o,[a_1,\ldots,a_j],k],IK) && (14)\\
(HP5) \quad = \quad & ma^1.\alpha(m,[o,[a_1,\ldots,a_j],k],IK) && (15)\\
(HM2) \quad \Rightarrow \quad & \mu(ma^1).\gamma(\alpha,ma^1).\alpha(ma^1,[m,[o,[a_1,\ldots,a_j],k],IK],IK) && (16)\\
(HP5) \quad = \quad & ma^2.\alpha(ma^1,[m,[o,[a_1,\ldots,a_j],k],IK],IK) && (17)\\
& \ldots &&\\
(HP5) \quad = \quad & BA.\alpha(ma^r,[ma^{r-1},[\ldots,[ma^1,[m,[o,[a_1,\ldots,a_j],k],IK],IK]\ldots],IK]) && (18)\\
(L2) \quad = \quad & ba(ma^r,ma^{r-1},[\ldots,[ma^1,[m,[o,[a_1,\ldots,a_j],k],IK],IK]\ldots],IK) && (19)
\end{aligned}
$$

Figure 5: General case for $o.s(a_1,\ldots,a_j)$ (apply phase).

the behavior of the system, let's again illustrate the general configuration of a method object `m`. As an object, `m` also has a meta-object, but this meta-object is mainly used to retrieve the apply method of `m` through the execution of a lookup method. Hence, the observations we made on the lookup phase also apply to the problem of finding the apply method of `m`.

But, once found, the apply method of `m` is also a method, having is own apply method and so on until `BA` is reached. Hence, there is a sequence of apply method objects ending with `BA`. The Figure 4 illustrates this construction. Again, to be able to follow the behavior of the system, we formalize this in the following problem hypothesis :

HP5 *The apply method of the method object m is ma ; ma has itself an apply method called* ma^1*, and so on for r levels where the rth level apply method has the basic apply method BA as its apply method, i.e.*

$$
\begin{aligned}
ma &= \mu(m).\gamma(\alpha,m)\\
ma^1 &= \mu(ma).\gamma(\alpha,ma)\\
ma^2 &= \mu(ma^1).\gamma(\alpha,ma^1)\\
\ldots &\quad \ldots\\
ma^r &= \mu(ma^{r-1}).\gamma(\alpha,ma^{r-1})\\
BA &= \mu(ma^r).\gamma(\alpha,ma^r)
\end{aligned}
$$

The Figure 5 develops the equations for the second phase of the message execution, namely the application of the method object `m` found during the lookup phase. Again in this case, the evaluator dives into the apply method sequence until it reaches the basic apply method that, by the evaluator hypothesis HE2, it executes using the primitive apply function to run the code of the previous method.

5.2 Discussion

Let's come back on each phases and assess their respective implementation. Let's also come back on the impact of model's and evaluator's hypothesis but also lemmas and theorems on the implementation of the language's evaluator.

5.2.1 Implementing the lookup phase

A careful examination leads us to split the lookup process in two subparts. First, the primitivity of the meta-object fetch makes the evaluator dive into the meta-object hierarchy down to `BMO`, as shown in lines (2) to (7). During this subpart, the evaluator accumulates apply continuations that makes the second subpart, which essentially climbs back from `BMO` to the meta-object of `o`, as shown in lines (8) to (12). At each step of this subpart, the lookup method of one meta-object is applied to find the lookup method of the following one, until it reaches the meta-object of `o` where it finds the method object `m`.

Although this lookup process can be very long, there is a large potential source for optimizations. 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 to each object upon the first message receiving and when changes are made to their respective meta-object hierarchy. Hence, the resulting efficiency can be near to the one of current object-oriented languages.

Notice, the great flexibility obtained by the use of explicit continuations. When providing a lookup method `L` to an object, it is possible, by supplying a specific apply method for `L`, to modify the current continuation of the message send operation. Normally, this continuation will contain a chain

of lookup method applications ending with the application of the method to answer the message itself. Hence, the user can deeply modify not only the lookup for an object but also the way it is applied and even the future computation. We will shortly discuss the efficiency problems appearing because of this flexibility.

5.2.2 Implementing the apply phase

During the apply phase, the evaluator accumulates the apply methods it encounters in the form of arguments to their next level apply method, leading to the expression $ba(ma^r, ma^{r-1}, [..., [ma^1, [m, [o, [a_1, \ldots, a_j], k], IK], IK]...], IK)$, which is directly executable under HE2. What does represent this expression? It executes the primitive apply function, which applies the method object ma^r to the object ma^{r-1}. But, ma^r is an apply method that executes the method object ma^{r-1}, which itself executes the method object ma^{r-2}, and so on until the method object `m` is executed on `o`. Each level has its own (default) continuation, which is initialized to the identity continuation but, thanks again to the flexibility of the continuation-passing style, that may be changed to fit applications' needs.

Obviously, we are in presence of a reflective tower such as the ones explicited by Smith [Smi84] : a tower of evaluators where the *nth* level evaluator executes the *(n-1)th* level evaluator and so on. Reflective towers are the cornerstone of behavioral reflection. 3-Lisp [dRS84] made them explicit, while 3-KRS [Mae87] gets its own and CLOS exhibits a similar characteristic [dR90]. So what is the difference between our approach and these?

First, the tower is finite and has a concrete representation in the language : apply method objects are the levels of the tower linked through their `applyTo:withArgs:withCont:` method slot. Second, and most important, towers appear in a method per method fashion, giving them statically a *degree of introspection* [dRS84], a number of levels they need to be run. Hence, we get a locality of effects insuring that only those methods that use a particular tower will be affected by the execution at the corresponding degree of introspection. The fact that we grasp the particular expression of the tower for one method at a time also suggests that optimizations may be easier. Third, climbing the tower is not made through reflective operations but by making changes directly on apply methods themselves. We claim that this is dual to the 3-Lisp approach.

Partial evaluation, semantics-based program transformations and advanced compilation techniques should be used to flatten the reflective tower associated with `m` in order to obtain more efficient equivalent code. Again, caching techniques can be used to keep track of these transformations and of the resulting optimized code from one application of a method to the other.

5.2.3 The evaluator

Some important clues about the implementation reside in the model and evaluator hypothesis, lemmas and theorems. Most model hypothesis are essentially constructive statements : they give an alternative view on the kernel of the model or they declare properties that must hold anywhere in a well-constructed application. This is the case of model hypothesis HM1, HM3, HM4, HM5, and HM6.

The hypothesis HM2, as well as evaluator hypothesis, have a different flavour. The reflective introspection rule is at the heart of the primitive evaluator behavior. How should the evaluator implement this introspection rule? As we have seen (§5.1), efficiency suggests that the eager development of reflective expressions is more amenable to optimizations. Hence, the primitive apply should not use it blindly, but rather implement an equivalent behavior, which would open the door to optimizations in much the same way as Kiczales suggested in [Kic90].

The second evaluator hypothesis, HE2 stops the introspection made by the reflective rule. Hence, it has a tremendous importance. The key idea of closing the infinite meta-regression at some definite point boils down in our model to the fact that `BA` represents the primitive apply's behavior. HE2 states the point where the execution can step from message passing among objects to the primitive apply, in a sort of dereification.

The solution to this, suggested by making `BA` its own apply method, is to have a primitive apply function *ba* written outside the object world and to code directly this dereification step, either by dynamically testing the hypothesis preconditions and switching levels appropriately, or by compiling methods in such a way that the system completely avoids sending apply messages to `BA` but executes the function *ba* in place. We also investigate other solutions.

The evaluator hypothesis HE3, about the representation of the method object `BL`, as well as lemmas, provide essentially opportunities for optimizations in the execution of the system. Having a fixed kernel would allow shortcuts to be used instead of repeatedly sending `lookup:in:` and `applyTo:withArgs:withCont:` messages among kernel objects. The same applies to Theorem 1 that generalizes all the preceding lemmas.

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* ∘ *apply* augmented with first-class continuations, is the cornerstone of our behavioral reflection model. An important characteristic of this model is the way it manages the infinite meta-regression of the *meta-of* link by supplying a basic meta-object that admits itself as its own meta-object.

We have given the necessary conditions to establish the correctness of this model. The main outcome of this exercise is that we obtain an abstract characterization of the conditions that must be respected by the language evaluator, which is independent of the exact representation of methods and of the implementation of the primitive apply function. Hence, it can be adapted to many different computing models.

We have then discussed the general behavior of the resulting language by examining step by step the execution of a message. Not only does this study gives a fine-grained understanding of behavioral reflection in our model, it also leads to two important conclusions.

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 [FJ89], this should be part of most object-oriented languages.

Second, the reification of the apply methods leads to a form of reflective towers *à la* 3-Lisp [dRS84], which con-

firms again the central role they are playing in behavioral reflection. Object orientation changes the perspectives by making towers local to each methods and our model made 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 the application of semantics-based program transformations as well as advanced compilation techniques to be adapted to reflective towers.

This is, in our view, the main research direction to be pursued in the near future in order to make behavioral reflection an effective tool. We are actually working on these problems and we are just beginning a large project to implement efficiently a new reflective object-oriented programming language based on this model. Indeed, much work has still to be done to obtain a system that would be able to cope with these ambitious implementation requirements.

References

[Bor86] A.H. Borning. Classes versus Prototypes in Object-Oriented Languages. In *Proc. of the IEEE/ACM Fall Joint Conference*, pages 36–40, 1986.

[Coi87] P. Cointe. Metaclasses are First Class: the ObjVLisp Model. *Proc. of OOPSLA'87, ACM Sigplan Notices*, 22(12):156–167, December 1987.

[Coi90] P. Cointe. The ClassTalk System: a Laboratory to Study Reflection in Smalltalk. In [WRM90].

[DMC92] 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.

[dR90] J. des Rivières. The Secret Tower of CLOS. In [WRM90].

[dRS84] J. des Rivieres and B. C. Smith. The implementation of procedurally reflective languages. In *Proc. of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 331–347, August 1984.

[Fer89] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. *Proc. of OOPSLA'89, ACM Sigplan Notices*, 24(10):317–326, October 1989.

[FJ89] B. Foote and R. E. Johnson. Reflective Facilities in Smalltalk-80. *Proc. of OOPSLA'89, ACM Sigplan Notices*, 24(10):327–335, October 1989.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 – The Language and its Implementation.* Addison-Wesley, 1983.

[HCCU90] 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.

[IEE91] IEEE, New-York. *IEEE Standard for the Scheme Programming Language*, IEEE Std 1178-1990 edition, 1991.

[JF92] S. Jefferson and D.P. Friedman. A Simple Reflective Interpreter. In [YS92], pages 48–58.

[Kic90] G. Kiczales. Making Reflection Safe for Real-World Users. In [WRM90].

[Mae87] P. Maes. Concepts and Experiments in Computational Reflection. *Proc. of OOPSLA'87, ACM Sigplan Notices*, 22(12):147–155, December 1987.

[MCD91] J. Malenfant, P. Cointe, and C. Dony. Reflection in Prototype-Based Object-Oriented Programming Languages. In [WRM91].

[MCD92] J. Malenfant, P. Cointe, and C. Dony. Behavioral Reflection in Prototype-Based Languages. In [YS92], pages 143–153.

[Smi84] 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.

[Ste90] G. Steele. *Common Lisp: the Language (2nd edition).* Digital Press, 1990.

[US87] D. Ungar and R. Smith. Self: The Power of Simplicity. *Proc. of OOPSLA'87, ACM Sigplan Notices*, 22(12):227–242, December 1987.

[WRM90] *Proc. of the First Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90*, October 1990.

[WRM91] *Proc. of the Second Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'91*, October 1991.

[WY88] 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.

[YS92] 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.