The OBJVLISP Model: Definition of a Uniform, Reflexive and Extensible Object Oriented Language

Jean-Pierre BRIOT*  Pierre COINTE†

Abstract

This paper presents the ObjVLisp model designed to experiment and synthesize the expanding activities of the Object Oriented world. The goal is to specify a minimal kernel whose semantic is perfectly uniform, then to modelize other Object semantics by extending this concentrate kernel.

We propose a unification of the metaclass, class and object concepts which allows: an optimal uniformity. This unification is obtained as a reflective definition of the language. We rapidly describe some ObjVLisp variations establishing the unification of the system.

The kernel is built on a portable virtual machine defined as a set of Lisp primitives which preserve the Lisp evaluator and maintain the spirit of dynamic (or lexical) scoping.

Keywords:

object, class, metaclass, uniformity, instance variable, class variable, message, instantiation, CLASS, inheritance, OBJECT, Lisp, Smalltalk, ObjVLisp, virtual machine, dynamic scoping, lexical scoping, portability, reflexivity, extensible.

1 Uniformity

"One way of seeing the Smalltalk philosophy is to choose a small number of general principles and apply them uniformly" [Kra83].

1.1 Is a class an object?

In most common Object Oriented Languages, besides Kraus's uniformity "wish", a class is not a REAL object. Some of them however, like Smalltalk-80 [Gra83] and Loops [BoS83], introduced the new metaclass concept to provide a greater abstraction. To definitely suppress the gap between class and object, we propose a unification of the metaclass, class and object concepts.

We claim [BoS85] that a class must be an object, allowing greater clarity and expressive power. As a consequence, we no longer need to introduce the class variable and class method concepts of Smalltalk which are not real variables and methods.

1.2 Class and terminal instance

The reverse question is "is every object a class?", whose answer is "no". Some objects are "only" instances of a class, and don't define a model. An instance of a POINT class, e.g. an object a:point, or an instance of the NUMBER class, e.g. 3, are such objects. We call them terminal instances.

1.3 How many object types?

We consider only one kind of objects, without structure or nature distinction between generators (classes) and non generators (terminal instances).

†Centre National d'Etudes des Télécommunications, B.P. 5049, 75705 Paris, electronic mail: javier@lis.fr, & LITP 6 place Jussieu, 75005 Paris.

References

In fact, they only differ by their capacity to react to the instantiation message. If the class of an object owns the primitive instantiation method (new selector, owned by the primitive class CLASS) or inherits it, this object is a generator, otherwise a terminal instance.

1.4 Metaclasses are usual classes

A metaclass (or a metagenerator) is simply a class which generates other classes. Every class declared as a subclass of the metaclass CLASS inherits its new method, and becomes a metaclass. Therefore the introduction of the metaclass concept is unnecessary and "the distinction between metaclasses, classes and terminal classes" is only an inheritance consequence and not a type distinction. There is now only one type of objects in the model. This unification simplifies the idea of instantiation and inheritance concepts but imposes to use them simultaneously: for example a metaclass is created as the subclass of another one (as an "ultimate" subclass of CLASS).

2 The ObjVlisp Model

Historically the ObjVlisp model comes from our work on Smalltalk-76 [Brie83], and our wish to present a synthesis in an operational semantic expressed in Lisp [Obi84]. We present here the new reflexive version which exactly integrates the previous unification and gives a fine solution to the problem of <class, instance> dichotomy.

2.1 ObjVlisp in five Postulates

The classical presentation of Smalltalk-76 [Gr78], five postulates fully describe the new ObjVlisp model:

P1: every entity of the language is an object.
    An object represents a piece of knowledge and a set of potentialities:

P2: the only control structure is message passing: the protocol to activate an object. A message specifies which procedure to apply (denoted by its name, the selector), and the arguments to pass:

    (send object selector Args, ... Args)

P3: every object belongs to a class that specifies its data (attributes called fields or instance variables) and its behavior (procedures called methods). Objects will be dynamically generated from this model, they will be called instances of classes. Following Plato, all instances of a class have same structure and shape, but differ through the values of their common instance variables.

P4: a class is also an object, generated from a class, called a metaclass (because describing a class). Consequently (P3), to each class is associated a metaclass. The initial primitive metaclass is the class CLASS.

P5: a class can be defined as a subclass of one (or many) other class(es). This subclassing mechanism allows inheritance of fields and methods, and is called inheritance. The class OBJECT represents the most common behavior shared by all objects.

2.2 Classes and objects

2.2.1 Structure of an object

The postulates P1 & P3 define an object as a "chunk" of knowledge and actions whose structure is defined by its class. More precisely:

fields: This environment (also called a dictionary) is split in two parts: a) the set of instance variables specified by the object's class, b) the set of associated values.

P2 of self and the three values associated to the CLASS's instance variables:

of self and the three values associated to the CLASS's instance variables:

Of course, the process for creating an instance of POINT using the same new message:

Then we create an instance of POINT, using the same new message:

The semantic and syntax of the new method are totally uniform. The new message always receives as arguments: the name of the instance to generate, and the values related to the instance variables specified by the receiver-class. Unlike the SMALLTALK-80 system, ObjVlisp uses only one method (named new) to create an object (by using the make-object primitive of the virtual machine). This method is owned by the metaclass CLASS as expressed by its reflexive definition.

3 Reflexivity

Because we need a complete transparency in the objects definitions to give more complete control to the users, we adapt the reflective interpreter technic [DRS84] to the construction of our model. ObjVlisp is supported by two graphs: the instantiation graph and the inheritance graph. The instantiation graph represents the "instance of" relation (P3 & P4), and the inheritance graph the "subclass of" one (P5). CLASS and OBJECT are the roots of these two graphs: they are defined in ObjVlisp.

This table shows the hierarchy of values owned by each object:

"draw" (point (by-pto x0 y0) x y)
3.1 CLASS: Instantiation

CLASS is the first object of the system, as the root of the instantiation graph, it will recursively generate all other objects. CLASS is also its own instance and we have the equality:

\[
\text{class [CLASS]} = \text{CLASS}
\]

3.1.1 Self pattern-matching of CLASS

To verify the previous equality we have to guarantee that the instance variables specified by CLASS match the corresponding values also held by CLASS, as its own instance:

\[
(\text{variables} : \text{supers} \quad \text{variables} \quad \text{methods})
\]

\[
(\text{values} : \{\text{OBJECT} : (\text{supers} \quad \text{variables} \quad \text{methods})\} \quad \text{new} \quad \text{lambda} \quad \text{menu})
\]

The value associated to the instance variable \text{i.variables} is exactly the list of instance variables itself, this self pattern-matching illustrates the definition of CLASS as an object.

3.1.2 Boot-strap

"A natural and fundamental question to ask, on learning of these incredibly interlocking pieces of software and hardware is: "how do they ever get started in the first place?", it is truly a baffling thing" [Kle79].

Defining CLASS from itself necessitates to precise the boot-strap mechanism. We create the skeleton of CLASS using the make-object primitive. We just need to introduce the new method which supports the self-instantiation of CLASS:

\[
\begin{align*}
(\text{make-object} \quad \text{<name>}) \to \text{<class>}) & \\quad \text{class} \to \langle\text{<name>}, \text{<class>}\rangle \\
\text{supers} & \\quad \text{variables} \quad \text{methods} \\
\text{new} & \quad \text{lambda} \quad \text{name} \quad \text{variables} \quad \text{values} \\
\text{make-object} & \quad \text{let} \quad \text{values} \quad \text{values}
\end{align*}
\]

This bootstrap prepared, we can define \text{ObjVlisp} in \text{ObjVlisp}.

3.1.3 Self-instantiation of CLASS

\[
\begin{align*}
(\text{send \ 'CLASS' \ 'new' \ 'CLASS'}) & \to \text{<name>}) \\
\text{NEW} & \quad \text{lambda} \quad \text{name} \quad \text{values} \\
\text{supers} & \quad \text{lambda} \quad \text{supers} \\
\text{variables} & \quad \text{lambda} \quad \text{variables} \\
\text{methods} & \quad \text{lambda} \quad \text{methods} \\
\text{methodfor} & \quad \text{lambda} \quad \text{methodfor} \\
\text{change} & \quad \text{lambda} \quad \text{change} \\
\text{understand} & \quad \text{lambda} \quad \text{understand} \\
\text{menu} & \quad \text{lambda} \quad \text{menu}
\end{align*}
\]

This definition is given for a dynamically scoped Lisp \text{LeLisp} [CO84] as other ones and examples. Then all the instance variables are automatically bound to their values in a method body. Consequently the lambda-methods associated to the supers, \text{i.variables} and methods selectors are quite easy to express.

Similarly, in the new method, \text{last} denotes the metagenerator (here CLASS), self is the generator (again CLASS), and \text{i.variables} the variables to instantiate with \text{i.values}. They are used as free variables in the make-object call.

3.2 OBJECT: Inheritance

(PS) precises the inheritance mechanism which concerns only classes. Inheritance allows to connect together fields and methods of several classes but in two different ways.

The inheritance of fields is static and done once at creation time.

When defining a class, its instance variables are calculated as the union of a copy of the instance variables owned by the superclass with the instance variables specified at creation (the value associated to \text{i.variables} in the new message).

On the contrary, the methods inheritance is dynamic and uses the virtual copy mechanism realized by the linkage of classes in the inheritance graph.

If the lookup of a method fails in the receiver class, then the search continues in a depth/breadth way. This graph is supported by the \text{supers} instance variable.

3.2.1 Classes vs Terminal Instances

The inheritance mechanism of fields is applied only when creating classes. Thus we need to discriminate creation of classes and creation of terminal instances. We have the equivalence:

\[
\text{create a class} \quad \text{<=> creator is a metaclass}
\]

As we pointed out already, a class inheriting from CLASS is a metaclass and inherits statically the \text{supers}, \text{i.variables} and methods instance variables. The predicate-method \text{metaclass?}, presented below, implements this test. It is used by the make-object primitive, thus no inheritance will occur when creating a terminal instance.

3.2.2 OBJECT the most common class

The second primitive class is \text{OBJECT}. Instance of \text{CLASS}, \text{OBJECT} is usually the default specified superclass (e.g. \text{CLASS} is a subclass of \text{OBJECT}), so it represents the most common class (intersection of all classes), describing the most common behavior (for classes and terminal instances).

\[
\begin{align*}
(\text{new} \quad \text{CLASS}) & \quad \text{<variables>}) \\
(\text{send \ 'CLASS' \ 'new' \ 'OBJECT'}) & \quad \text{<values>}) \quad \text{<values>}) \\
\text{<classname>} & \quad \text{lambda} \quad \text{i.variables} \\
\text{<value>} & \quad \text{lambda} \quad \text{values} \\
\text{<class>} & \quad \text{lambda} \quad \text{class} \\
\text{print} & \quad \text{lambda} \quad \text{print} \\
\text{error} & \quad \text{lambda} \quad \text{error}
\end{align*}
\]

From this definition, each \text{ObjVlisp} object answers to the \text{<selector>} by \text{<action>}:

\[
\begin{align*}
\text{is} & \quad \text{going} \quad \text{the} \quad \text{name} \quad \text{of} \quad \text{this} \quad \text{class} \\
\text{?} & \quad \text{returning} \quad \text{the} \quad \text{value} \quad \text{of} \quad \text{the} \\
\text{?<} & \quad \text{writing} \quad \text{the} \quad \text{value} \quad \text{of} \quad \text{the} \\
\text{variable} & \quad \text{returning} \quad \text{the} \quad \text{values} \quad \text{of} \quad \text{the} \\
\text{class} & \quad \text{testing} \quad \text{if} \quad \text{the} \quad \text{object} \quad \text{is} \quad \text{a} \quad \text{metaclass} \\
\text{print} & \quad \text{printing} \quad \text{an} \quad \text{external} \quad \text{representation} \\
\text{error} & \quad \text{precising} \quad \text{the} \quad \text{standard} \quad \text{error} \quad \text{treatment}
\end{align*}
\]

Notice that the \text{?} and \text{?<}-methods which access to the instance variables are conform to the dynamic binding of \text{LeLisp}.

3.3 Self-Extensions

3.3.1 Class variables by Example

Let us return to the POINT class, previously defined. Now we would like the constant character \text{~*} to be a class variable shared by all the points of a same class. We redefine the POINT class as before, but metaclass of which (let's call it \text{METAPoint}) specifies this common character:
3.3.2 Architecture of the ObjVlisp model

We summarize the general structure of the ObjVlisp model by connecting together the instantiation graph and the inheritance graph to explain the "POINT" construction:

![Diagram showing the architecture of ObjVlisp model]

METAPONT is declared as a subclass of CLASS (thus it is a metaclass). It inherits the instance variables super, 'variables and methods from CLASS and adds them the instance variable char. Consequently, POINT specifies the associate value of char, i.e. 'a'..

Now we could create such a point:

```lisp
? (send 'POINT new 'a' point 20 't')
+a:point
? (send 'a's point draw)
*+
```

Parametrization of a class: The POINT class is now parametrized by the "display" character and the METAPONT metaclass represents this abstraction. Let's define two new classes, called POINT# and POINT@ with two other different display characters. Obviously, they are defined as a subclass of POINT:

```lisp
? (send 'METAPONT new 'point Point 0 0 't')
+POINT
? (send 'METAPONT new 'point@ Point 0 0 's')
+POINT@
? (send 'POINT new 'a's point 1 2) draw
#
? (send 'POINT@ new 'a's point 3 4) draw
@
```

Such a simple and intuitive construction is IMPOSSIBLE in Smalltalk, because class variables are implemented like pool variables [GRS], and are not inheritable.

With the ObjVlisp model we have the identification:

```
class variable [an_object] =
instance variable [an_object's class]
```

3.3.3 Class methods by Example

As for class variables, class methods are specified in the metaclass as usual methods. Suppose we want to define a new class method for POINT to create and initialize a new point. We simply define the new-init method of METAPONT (assuming we define also an init method in the POINT class, or at least in the OBJECT class):

```lisp
? (send 'METAPONT understand new-init
  (send 'self new 'name init))
```

3.3.4 Filiation link

To use classes registering all their instances, we define a new metaclass SET, as a subclass of CLASS with a new instance variable zone pointing the list of instances. We just have to redefine the CLASS new method to add the new instance at the end of the zones list:

```lisp
? (send 'CLASS new 'SET (CLASS zones)
  (send 'self new 'name new-value init)
  (send 'self new 'variables set
    (send 'self new 'values self)
    (send 'self new 'set-metaclass))
```

The mappings method distributes an unary message (without argument) to all the instances of a particular class. The "(run-supers)" call (same as in CommonLisp [BKG*83]) recalls the current message, but the lookup for the method will begin in the superclasses of the current class.

4 Implementation

ObjVlisp means Object extension of a Virtual Lisp and is implemented along the virtual machine technic allowing a quick instantiation on any Lisp (dynamically or lexically scoped). The classical difficulty when defining such a machine is to choose general primitives available on major aimed systems without privileging particular constructions, even if they are powerful for a given target. Consequently we have decided to preserve the <eval>, apply > engine, and use classical Lisp data structures such as atom, list, structure or environment. The Lisp instantiation of this virtually machine is totally described in [CoBa84], and we are just finishing its Scheme and CommonLisp mergings.

To synthesize our presentation we conclude this paper with the ObjLisp & Scheme definitions of an object:

```
ObjLisp : an object is implemented as a Lisp atom whose functional value (send is equal to the funcall function) is a self-reference.
```

1. The pseudo instance variables are dynamically bound by the "setf" function. Notice that those variables allow a factorization of the script of an object, the parts -to- and a defining a completely parametrized (and shared) piece of code.

2. In the spirit of a dynamic Lisp, the activation of an object is done in its context. This double environment is dynamically set by the two "letf" (let dictionary) functions:

   "letf (variables metaclass) (variables list)"

   sets the class environment, and

   "letf (variables list) (variables self)"

   the object's one,

3. When the object is deactivated, the updating of the double closure is automatically done with the function "revert". Consequently, the usual Lisp affectionation functions (seti, setq, next, incr...) can be used to assign the instance (and class) variables.

ObjScheme: an object is now anonymous and represented as a compound-procedure grouping together the environment of the instance variables (i.e. list) and a lambda-expression:

```
:obj = (cons (class) (ten (ten (ten ...)
```

- apply (lookup selector (access list) (i.e. list))
- (cons) (self selector arg)

6We propose at ObjScheme) to replace these "letf" pseudo-variables bindings by real instance variables, defining OBJECT with init and metaeq (bijective) as instance variables. Objects can also be anonymous by using structures in place of atoms (notice that naming classes is useful for documentation and debugging, therefore CLASS will own the new instance variable).
5 Improvements and Possibilities

The Obj/Visp model first advantage is UNIFORMITY. There is now only one kind of objects. This allows a simplification and reduction of concepts, which are thus more powerful and general. The second property is REFLEXIVITY which provides a language totally and uniformly accessible by the user. Finally, EXTENSIBILITY authorizes various applications and semantics modelizations. For example the study of inheritance strategies of such systems like Flavors [MV86], Smalltalk-80 [GR83] or Loops (BS83) are simulated by defining new metaclasses [Coi84].

The Obj/Visp project is part of the "O.O.P. Methodology GRECO de Programmation" Research Group.

References


MULTILOG : MULTIPLE worlds in LOGic programming

Hervé Kauffmann
Laboratoires de Marcoussis
C.G.E. Research Center
Route de Noisy
91460 Marcoussis
France

Alain Grumbach
Ecole Supérieure d’Electricité
Plaute du Moulin
91190 Gif
France

ABSTRACT

MULTILOG is a Logic Programming language intended for knowledge representation and manipulation. Its main features are the following:

- knowledge is distributed among different worlds;
- each world has its own inference mechanism;
- several inheritance relations are provided.

Possible applications of MULTILOG include: hypothetic reasoning, default reasoning, viewpoints representation, distributed reasoning.

1. INTRODUCTION

Our purpose is to use logic programming to represent and manipulate knowledge. Unfortunately, there are features essential to knowledge representation that are difficult to handle if we use a logic programming language such as Prolog (<Colmerauer 79>), <Kowaliski 79>:

- Representing and manipulating large amounts of knowledge.
  - example : If we want to perform symbolic integration, we need a lot of knowledge ; general knowledge on integration (e.g. the integral of a sum is the sum of the integrals), as well as specific knowledge on rational functions, e.g. how to use the method of partial fractions, on trigonometric expressions (e.g. which are the interesting substitutions such as u = tan(2x)), etc.

This knowledge needs some structure to achieve clarity and efficiency.

- Default reasoning.
  - example : We often have to represent facts such as:
    - If you do not know where somebody lives, then it is reasonable to assume that he lives where his work is located.
    - Typically, a mammal has four legs.
  - Such facts can be expressed in Prolog only through using such extra features as meta-level features (e.g. "clause" built-in predicates) or control facilities (e.g. "if").

- Hypothetical reasoning.
  - example : Suppose that we wish to perform troubleshooting on logic circuits. To show that a given component may be responsible for the failure, we hypothesize the faultiness of this component and then check to see whether the circuit outputs observed match the outputs expected.

It is possible to do this kind of reasoning in Prolog, but side effects built-in predicates such as "assert" and "correct" must be used. Besides, if we want to compare the consequences of two different hypothesis, the problem becomes much more complicated.