Reflexive Architecture: From ObjVLisp to CLOS

Nicolas Graube*

Equipe Mixte L.I.T.P. & Rank Xerox France
Université Paris VI
Tour 45-55 Porte 209
4, Place Jussieu
75252 Paris cedex 05
France

Abstract

This paper presents the design of a minimal set of instruction for a class system embedded in Lisp: ObjVLisp. We re-use the set of postulates describing the operational behaviour of ObjVLisp to discuss and derive a new implementation based on a reduced set of functions and a more self-contained description. Then we develop the ObjVLisp experience in building metaclass architecture to propose a layered and incremental meta-object protocol for CLOS.

1 Introduction.

One goal of the ObjVlisp project is to provide the user with a class system both understandable, self-described, small (described in approximately two pages of Lisp) and extensible to other different class system paradigms.

*This research was partly funded by the *GRECO de programmation du CNRS*.

To reach this goal ObjVLisp is constructed with a virtual machine[1] (a set of primitive functions [7]), and a circular definition of the basic class architecture (`Class` and `Object`).

This paper attempts to discuss, clarify and minimise this virtual machine. This minimisation will be operated mainly by reinforcing the self-description, i.e. by moving knowledge from the virtual machine to ObjVLisp itself: the object world.

Then we focus our attention on Common Lisp Object System (CLOS)[2], an object-oriented paradigm for Common Lisp [11]. CLOS enhances fusion into Common Lisp through both syntactical and semantical choices. These choices are synthesized in integration of message passing in functional application via generic functions and merging structure and Common Lisp types in CLOS' classes.

With the previous virtual machine, modified in order to characterize the behaviour of CLOS, we also attempt to describe a minimal virtual machine and a layered metaclass architecture for CLOS.

Since our main concern is minimality and reflexion, we do not address the efficiency problem even if both concerns are not antinomical. Optimizations are beyond the scope of this paper.

2 ObjVLisp's Postulates.

ObjVLisp is a good synthesis of a class system based on six postulates which both fully depict the operational behaviour of the system and give a guideline to possible implementations. These postulates were descibed in [7] as follows:

P1 : An object represents a piece of knowledge and a set of capabilities.

P2 : The only protocol for activating an object is message passing: a message specifies which procedure to apply (denoted by its name, the selector), and its arguments.

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

P4 : A class is also an object, instantiated by another class: its metaclass. Consequently (**P3**), to each class is associated a metaclass which describes its behaviour as an object. The initial primitive metaclass is the class `Class`.

[1] ObjVLisp stands for Object Virtual Lisp and does not rely on any particular dialect of Lisp. Different implementations have been provided for Le_Lisp (A., Deutsch, J-P., Briot), InterLisp-D (N., Graube), (Kyoto, Xerox) CommonLisp (A., Deutsch & C., Consel, N., Graube) and VLisp (P., Cointe)[4] [7].

P5 : A class can be defined as a subclass of one (or many) other class(es). This subclassing mechanism allows sharing of instance variables and methods, and is called inheritance. The class `Object` represents the most common behaviour shared by all objects.

P6 : If the instance variables owned by an object define a local environment, there are also class variables defining a global environment shared by all the instances of a same class. These class variables are defined at the metaclass level.

3 Discussion of the ObjVLisp Implementation.

We will now describe a new implementation of ObjVLisp where we use postulates as a guideline for the discussion of the various components of both the virtual machine and the object world. When necessary we will pinpoint some possible alternatives to the interpretations of postulates.

The first point to define is the basic class architecture as we need it in the following explanation. The architecture is defined by **P4**. It implies the presence of a self-instantiated class, `Class`, which will be the root of the instantiation tree. Once the self-instantiation problem has been solved the possibility of self description arises, then `Class` defines its own structure in such a way that it becomes both its class and its instance. We call this architecture a reflexive descriptive architecture, because while we share the same fundamental idea of meta-description with reflective languages[9], the word "descriptive" is used to express that we are only concerned with the description of the architecture, not with the operational process of *reflection.*

We use the reflexivity of the system for translating some of the instructions of the previous virtual machine[7] into the object world. Thus a common problem is infinite regression triggered by an over powering self-description.

3.1 The Link Between an Instance and its Class: Isit.

P3 represents an important problem. An object cannot live without its class, otherwise it cannot find the real associated semantic of all its embeded values.

Then a problem with class languages is to provide a way of representing the class of any object.

We have to decide the real status of the class reference. Should it be hidden from the user, thus part of the virtual machine, or accessible and thus part of the object world?

While the first proposition enhances safety and allows modifications only to be accomplished at the lower level, the second one enforces reflexivity and thus authorises any kind of manipulation at the object level.

The solution adopted by ObjVLisp is to put the class reference into an instance variable named `isit`[2] defined by `Object` thus, according to **P5**, inherited by every object of the system.

Once the status problem has been solved we need to describe the process involved in the search for this information. As `isit` is an instance variable, we could provide a method to access its value, triggered by message passing, which would obviously be defined in `Object`. Unfortunately this method leads us into an infinite regression because, in order to apply a method, along the lines of **P3** we must find the object's class, precisely what this method supposedly returns.

Then a method cannot be of any help in solving this problem, we have to define a specific function, not part of the object world, for retrieving this information. This function, `Class-of`, must hold some knowledge about where it can find the link in any object between itself and its class. This implies that every object will share the same basic shape, usable by `Class-of`.

3.1.1 The Basic Object Creation: `Make-Object`.

Defining `Class-of` means freezing both the global shape of every object and the physical position of the class reference within this structure. Then it has to be defined in conjunction with the basic object allocator, `make-object`, which will be the only entry-point for memory allocation[3].

```
(defun make-object (a-class)
       (let ((an-object (make-sequence 'vector 2)))
            (setf (first an-object) a-class) an-object) )

(defun class-of (an-object) (first an-object) )
```

3.1.2 Named vs Anonymous Classes.

Another point to sort out is the kind of information we expect to find in the link. Two sorts of information can be provided: the name of the mother class or a reference to the mother class.

---

[2]The name `isit` was chosen for historical reasons in relation to Smalltalk-72.

[3]For practical reasons, `make-object` will also set the link between the newly created object and its class i.e. the caller of `make-object`.

Storing the name of the class implies that the system can only handle named classes and thus reduce the expression power of the extension. However this scheme enables us to provide a very simple self-description for `Class` because defining a new class with the same name as an already existent one will automatically erase the latter.

On the other hand, storing the reference of the mother class frees us from the naming problem and enables us to use anonymous classes. While with the class name the self-description was trivially easy, this process needs to be more intricate and involves a pointer manipulation and more precise knowledge of the object structure.

3.2 Object Creation: new.

From **P3** we learn that instances are dynamically generated from the class. In combination with **P2** we know that creation is triggered through message passing and that the creation method must be located in the class of the class from which we want to make a new instance. Thus it is in the metaclass that we will find the description of the instance generator. This method will give the physical description of the instance which must be coherent with the general shape explained in the previous section, i.e. it must rely upon the use of `make-object`.

While the global shape cannot be customised because of `class-of`, the inside organisation of the new instance, i.e. where we put the associated value of the instance variables, is under the control of the creation method.

Three different schemes can be derived, each one defining different possibilities of parametrisation.

1. As generating a new object is performed by a method, we can include in it the description of the parametrisation or export it in an other method as follows:

```
new       (λ (self  &rest args)
;; self is automaticaly bind to the object receiver.
             (send  (send self 'basicNew (iv-value self 'iv)) 'initialize args) )
basicNew  (λ (self  ivs)
             (let ( (an-object (make-object self))
;; This is the parametrisation.
                    (structure (make-sequence 'list (length ivs) )) )
;; Here is an assumption about the global shape of an object, i.e.  a two field sequence.
                    (setf (second an-object) structure) an-object) )
```

This first solution implies that we hold some methods which give information about this parametrisation essentially needed for retrieving values.

2. Unlike the previous solution we use an instance variable of the metaclass to store the description of the parametrisation.

```
new  (λ (self &rest args)
        (send (funcall (iv-value (class-of self) 'constructor) self (iv-value self 'iv))
              'initialize args) )
```

Then all instances of classes of the same metaclass will share the same internal structure.

3. With the same idea as the previous solution, but by storing the value of the instance variable in the class, we get the following description:

```
new  (λ (self &rest args)
        (send (funcall (iv-value self 'constructor) self (iv-value self 'iv))
              'initialize args) )
```

Allowing different classes of the same metaclass to customise their instances in a different manner.

In both of the latter solutions, `constructor` can be associated to:

```
#'(λ (self ivs)
     (let ((an-object (make-object self))(struct (make-sequence 'list (length ivs))))
          (setf (second an-object) struct) self))
```

We have derived three different correct definitions for the method `new`, which is part of the object world, but all of them must depend upon the use of the allocation primitive strictly derived from the postulates.

3.3 Finding the value of instance variables: iv-value.

One fundamental action is to ask for the associated value of an instance variable inside an object[4]. As the Object world cannot handle the basic access to the associated instance variable[5], there must be a primitive of the virtual machine to perform this task.

[4]In the previous code this was accomplished by the Lisp form: `iv-value`.

[5]Providing a method for this task will cause infinite regression because of method description under instance variables, or implies another primitive for accessing the specific value associated to the instance variable holding the method description.

3.3.1 The Quest for Meaning.

From **P3** we learn that all of the description of the data of any object is held by its class. Therefore there must be some information in the class about instance variables defined for its instances.

In ObjVLisp this knowledge is described in the `iv` instance variable which holds all instance variables' specific information, i.e. their names, initial values and positions in the related object. Finding the associated value of a specific instance variable of an object depends on finding the associated value of the `iv` instance variable in an object's class and extracting all the information needed for retrieving the value. In order to complete this task we must be able to find the associated value of `iv` in the object's class and so on from class to class class. But from **P2** we know that there is a loop in the instantiation graph and while all classes share this graph, the latter process will loop endlessly. Thus, in order to avoid this infinite regression we have to provide some information at a certain point in the search. As the loop starts at `Class`, we must provide the description of all instance variables defined by it. This will be done via a primitive instruction from the virtual machine: `class-all-iv` which takes an object supposed to be the self-instantiated class and returns the description of all its instance variables. Then a possible description of this function is:

```
(defun iv-value (an-object a-name )
      (find-slota-name an-object
                (if (eq an-object (class-of an-object)) (class-all-iv an-object)
                    (iv-value (class-of an-object) 'iv) ) ) )
```

In which *find-slot* is only a do-loop which finds the named *a-name* instance variable description through the list of all instance variable descriptions, and then finds the associated value in the object *an-object*.

`Class-all-iv` is very simple to describe but relies on the internal organisation of the class `Class`.

3.3.2 The Quest for Physical Structure.

Because only the general shape is defined as being the same for each object, it is at the metaclass level, or at the class level, depending on the choice of implementation, that we can find the information about the internal physical structure of an object. Therefore the previous section only defines the way to retrieve the description of a specific instance variable. We have to modify the previous definition so that it will also find the physical description of the object in order to access the associated value embedded in this structure. As before we have to provide a primitive instruction, `class-organization`, which will give some information about the structure of `Class`, to avoid infinite regression[6].

[6]All `iv-value` description must be modified if we want to handle the `isit` case properly. But in order to clarify the presentation we omitted this treatment.

```
(defun iv-value (an-object a-name )
      (multiple-value-bind (iv structure)
                (if (eq an-object (class-of an-object))
                    (values (class-all-iv an-object) (class-organization an-object))
                    (values (iv-value (class-of an-object) 'iv)
                    (iv-value (class-of (class-of an-object)) 'structure) ) )
                (find-slot a-name an-object iv structure) ) )
```

3.4 Handling Multiple Inheritance.

P5 induces the necessity of handling multiple inheritance schemes. This can be part of the object world as the only implication is the fact that the path produced by this scheme can be found both by the control structure, while finding the description of the method, and during the collecting of all the inherited instance variables. Thus we can define a method located in `Class` for computing a linearisation of the multiple inheritance graph i.e. also named linear extension. This facility enables us to provide different inheritance schemes coexisting within the same system.

3.5 Control structures: send and run-super.

According to **P2** there is only one control structure in the object world which is **send**. While finding the right method can be easily parameterised via the contents of the instance variable which holds the linear extension and is deduced by the multiple inheritance treatment, the application of this method onto the arguments is not part of the object world. Then **send** has to be part of the virtual machine.

```
(defun send (an-object a-selector &rest args)
      (apply (find-method a-selector (iv-value (class-of an-object) 'linear-extension) ) )
            an-object args) )
```

A problem related to **send** is the processing of shared definitions of methods via the **run-super**. As we use multiple inheritance, **run-super** can no longer be just a static reference to the first super class of the class definition (as in Smalltalk-80[8]), handled by some processes which are part of the object world, i.e. *code walker* or others, but must be involved in a more general dynamic process, and must be described as a primitive. This also implies that some information has to be provided by `find-method` to permit this behaviour. Thus finding the method must also return the class of the description in order to go further in its search when executing a **run-super**.

Therefore we summarise the definition of a reduced virtual machine for ObjVLisp in the following figure 1.

class-of	make-object	iv-value	setf-iv-value
send	run-super	class-all-iv	class-organisation

Figure 1: ObjVLisp Reduced Instruction Set.

We made a first attempt at the description of a meta-object protocol for CLOS by using ObjVLisp's ability to describe new class taxonomy paradigms. Then we defined CLOS as an embedded world of ObjVLisp and we provided a set of classes which depicted all specific CLOS behaviour. More precisely we concentrated all of the differences in a metaclass which made the connection between the two worlds. Unfortunately even if this scheme works, the cooperation between these two worlds was unsatisfactory because some basic CLOS principles go against some basic ObjVLisp's ones[7].

Then while this experience was not, in our opinion, a complete success we decided to apply the same description process as the one for ObjVLisp to CLOS.

4 Common Lisp Object System in Eight Postulates.

Surprising as it might be, ObjVLisp and CLOS are depicted by some common postulates. Therefore we shall only pointout the differences.

P2' : The only protocol for activating an object is generic function application. A generic function is made up of different methods described for specific typed parameters. When applying a generic function the system tries to find the best defined method of this generic function, i.e. the one for which specific parameters are closer to actual arguments according to class inheritance.

P5' : A class can be defined as a subclass of one (or many) other class(es). This subclassing mechanism allows sharing of instance variables and methods, and is called inheritance. The class T represents the most common behaviour shared by all objects and the least defined type of Common Lisp.

P6' : If the instance variables owned by an object define a local environment, there are also class variables defining a global environment shared by all the instances of a same class. These class variables are specific instance variables defined in the class but with a particular allocation scheme. We call them *slots* instance variables or class variables.

P7 : As long as it is possible, elements of the extension such as slots, methods and generic functions are to be considered as objects and instances of specific classes.

[7]This needs to be synthesized in major modifications of the virtual machine.

P8 : Inside methods we can use inheritance via `call-next-method`. But in an orthogonal way we can use *method combination* which enables us to organize methods in a specific way.

5 Our Implementation of CLOS.

As for ObjVLisp an implementation can be depicted using postulates as guideline. In this case we do not give all the details but only focus on the major differences and on the associated reflexive descriptive architecture[8].

Note that some specific CLOS behaviour is not defined via postulates such as temporized instantiation, class changing protocol or specific method combination, mainly because these are not a fundamental aspect of CLOS behaviour but rather features which can be defined in the object world.

5.1 Basic access to Slots.

P6' and **P7** introduce important changes and imply both a generalised vision of instance variables and the consideration of some new primitive functions.

First we have to modify the `iv-value` so that access to both true instance variables and class variables is possible. In order to acces class allocated instance variables one solution is to describe these objects as instances of a specific class, `class-allocated-slot`, and instance allocated instance variables as instances of `instance-allocated-slot`. Then when we try to find the associated value of a slot in an object, we just have to apply a generic function, `get-value`, defined on both the instance and the slot description. Providing two different methods, one for each slot class, can solve the problem of where to find the associated value. Unfortunately this pretty solution leads once again to infinite regression because every class in the system describes their slots as objects, particulary generic function objects. Then we have to rely on a simple test on the class of slot description or simply on some information embedded in the description.

Secondly we have to consider the fact that slots are now true objects and that the simple action of finding the name of a slot in a slot description is a real problem. As applying a generic function leads us to infinite regression we have to provide two simple functions: `slot-name` and `slot-description`.

[8]Roughly all the modifications can be made by considering message passing as an application of a generic function where only the first argument is used in the method selection.

As there are no other great differences from ObjVLisp virtual machine description, we can summarise all the instructions in figure 2. Note that basic differences are limited to slots and the control structure otherwise they share the same description.

class-of	make-object	slot-value
setf-slot-value	slot-name	slot-description
class-all-slots	class-organisation	call-next-method

Figure 2: CLOS Instruction Set.

6 A Layered Architecture for CLOS.

While the external description of CLOS object behaviour can easily be found in the first two chapters of [2], it is not the case with the meta-object protocol which is still under discussion. However we try remain as faithful as possible to the first ideas explained in [1].

A basic principle is to define a reduced set of classes, thus defining a kind of *micro*-CLOS (μCLOS), which describes all basic CLOS behaviour. Then, by using μCLOS as a basis we will build all specific CLOS classes. This stratified construction will be completed using multiple bootstrapping as some classes are mutually defined. The key notion is that from bootstrap to bootstrap we get closer and closer to the final description of CLOS, coherent with all postulates. Then it is not surprising to give an incremental description of the virtual machine as well. Each description must be coherent with the postulates which have been introduced by the previous bootstrap[9].

6.1 The ObjVLisp Kernel Revisited: μCLOS.

Along the lines of **P3** we can define the very first kernel of μCLOS: the self-instantiated class `Class`, root of the instantiation tree and its first instance from which it inherits, the class `Object`. While `T` has been defined as being the most common class of all the components of the CLOS system, it is more like the only connection with the Common Lisp type system than `Object` which has to be considered as the most common class of all CLOS classes.

[9]As ObjVLisp was described with a single bootstrap this necessity did not appear in its description.

Defining `Class` and `Object` will need a first bootstrap and the use of an early version of `make-instance`[10] and of `defclass`. As generic functions cannot yet be provided `make-instance` is only a simple Lisp function using `make-object`.

6.2 `Class`, `Object` and `Slot`: The Holy Trinity.

P7 implies that slot descriptions must be instances of a class. We can then instantiate a new object from `Class`: the class `Slot`. It inherits from `Object` and depicts all the behaviour for slot description. While all existing slot description must be instances of this class, we must perform a second bootstrap in order to give a coherent definition to `Object`, `Class` and `Slot`.

`Slot-value` needs to be modified in order to handle object slot description. To be more precise we just have to give the final description of `slot-name` and `slot-allocation` while their corresponding postulate is introduced into the system. We have to rewrite some of the `defclass` description to make it handle slot instantiation and to provide a more complete definition of the `initialize` function so that it distinguishes between terminal objects and classes.

6.3 Types and `T`: A Connection with the Type System.

`T` is an entry point in the Common Lisp Type Lattice but is only there to be inherited and must act like an abstract class. However as Common Lisp Types have to be considered as classes we must provide a hook where we can describe their representation in terms of class. Thus we have to provide a metaclass `Types`, instance of `Class`, from which we describe all the necessary classes.

If the `Types` definition does not involve any particular problem it is not the same for `T` as it must be inherited by `Object`. Thus a third bootstrap is necessary enabling us to provide the right description of `Object`.

6.4 `Methods` and `Generic-Functions`.

Each time we had to check the class, or the type of an object to specify the correct behaviour of a function, we performed an explicit test, embedded in this Lisp definition. We will now introduce all necessary classes in order to integrate **P2'**. Two classes can describe basic generic functions, i.e. generic functions which handle both method definition on a cartesian product of

[10] `make-instance` is ObjVLisp's new correspondant for CLOS.

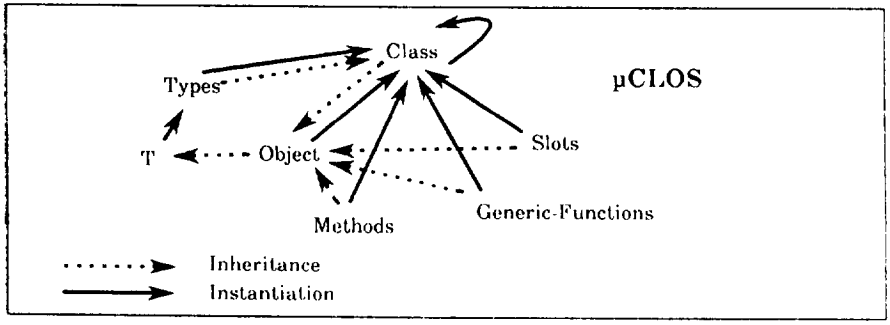

Figure 3: μCLOS.

classes or types and use of the hierarchy defined on classes or types and the code sharing scheme via the `call-next-method` utilisation. These classes are `Methods` and `Generic-Functions`, their definitions are straightforward and do not need any other bootstrapping mechanism.

Once they are defined we can move all the Lisp functions defined on objects into generic functions. At this point μCLOS is complete, it defines both a full class system for Common Lisp and a good basis for an extented description of CLOS.

6.5 CLOS as an embedded world of μCLOS.

When we made a description of CLOS as an embedded world of ObjVLisp, we encountered some problems as some of ObjVLisp's postulates clash with some basic CLOS ideas.

Describing CLOS as an embedded world of μCLOS while sharing the same idea as the previous attempt does not cause any difficulties because none of μCLOS's postulates can differ from CLOS ideas.

Our only task is to extend μCLOS classes so that they will describe the complete behaviour of CLOS. We started this process by increasing `Class` and `Slot` so that we have complete slot description instances of `Clos-Slot` which inherits from `Slot` and is co-instantiated with `Clos-Class`, a subclass of `Class`. This description needs an ultimate bootstrap as a result of the co-instantiation process. Note that as this bootstrap only concerns these two recently defined classes, the redefinition can only be local. A complete CLOS system has been implemented along this line and runs under Xerox Common Lisp. Finally because CLOS's generic functions are more powerful than the one μCLOS defines we must describe subclasses

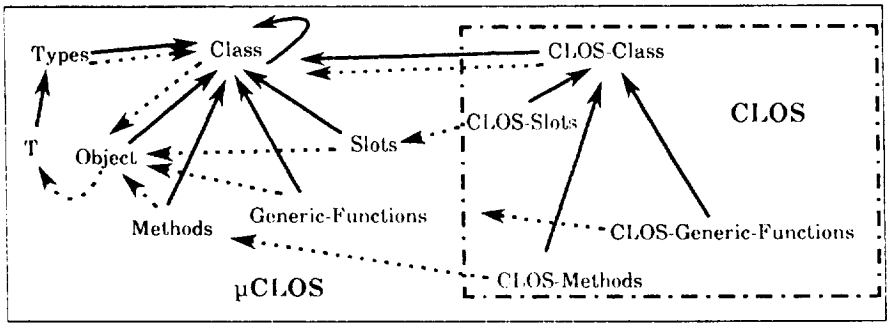

Figure 4: CLOS as embedded world of μCLOS.

of `Methods` and `Generic-Functions` respectively as `Clos-Methods` and `Clos-Generic-Functions` in order to depict their full behaviour.

7 Future Work.

The descriptions of the ObjVLisp and CLOS virtual machines were accomplished using postulates as a guideline. A second step which needs to be carried out by is an automatic derivation of a virtual machine through postulates, using denotational semantics or other formal descriptions. In the same direction as the one taken for the description of both ObjVLisp and CLOS, we will describe a complete class system for Le_Lisp Version 16[5]. This system will enhance metaclass programming and complete integration of the type system which will be defined at the same time.

Acknowledgements

We would like to thank Jean-Pierre Briot, Pierre Cointe, Alain Deutsch, Sara Dootson, Jean-François Perrot and Christian Queinnec for their encouragement and constant help.

References

[1] Bobrow, D.G., DeMichiel L.G., Gabriel R.P., Keene S., Kiczales G., Moon D.A, Common Lisp Object System Specification, Chapter 1, 2 and 3, X3J13 (ANSI COMMON LISP), March 1987.

[2] Bobrow, D.G., DeMichiel L.G., Gabriel R.P., Keene S., Kiczales G., Moon D.A, Common Lisp Object System Specification, Chapter 1 and 2, X3J13 (ANSI COMMON LISP), November 1987.

[3] Bobrow, D.G., Kiczales G., The Common Lisp Object System Metaobject Kernel A Status Report, *IWoLES 88*, Afcet, Afnor, LITP and Inria, Paris, France, February 1988.

[4] Briot, J-P., Cointe, P., A Uniform Model for Object-Oriented Languages Using The Class Abstraction, *IJCAI 87* Proceedings of the Tenth International Joint Conference on Artificial Intellingence, pp. 40-43, Milan, Italy, August 1987.

[5] J., Chailloux, M., Devin, J-M., Hullot, LeLisp, a Portable and Efficient Lisp System, *Lisp and Functional Programming*, pp 113-122, Austin, Texas, USA, August 1984.

[6] Cointe, P., Towards the design of a CLOS Metaobject Kernel: ObjVLisp as a first layer, *IWoLES 88*, Afcet, Afnor, LITP and Inria, Paris, France, February 1988.

[7] Cointe, P., Metaclasses are First Class: the ObjVlisp model, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pp. 156-167, Orlando, Florida, USA October 87.

[8] Goldberg, A., Robson, D., Smalltalk-80 - The Language and its Implementation, Addison-Wesley, Reading MA, USA, 1983.

[9] Maes, P., Concepts and Experiments in Computational Reflection, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pp. 147-155, Orlando, Florida, USA October 87.

[10] Queinnec, C., Cointe P., Types, Classes, Metatypes, Metatypes Classes: an open-ended data representation model for Eu-Lisp *to appear in the Lisp and Functional Programming conference*, Paris, LITP, January 88.

[11] Steele Jr., G., F., Common Lisp: The Language, Digital Press, 1984.

A ObjVLisp Reduced Instruction Set.

```
;;; A Reduced Instruction Set for ObjVLisp.
;;; Object allocation.  [Primitive 1]
(defun make-object (a-class)
     (let ((basic-structure (make-sequence 'vector  2)) )
        (setf (elt basic-structure 0) a-class)
        basic-structure) )
;;; Find the link between an object and its class.  [Primitive 2]
(defun class-of (an-object)  (elt an-object 0))
;;; Message Passing.  [Primitive 3]
(defun send (an-object selector &rest args)
    (do ((classes (iv-value (class-of an-object) 'linear-extension) (rest classes)))
        ((null classes) (send an-object 'error "Method ~S not found" selector args))
        (do ((methods (iv-value (first classes) 'methods) (rest (rest methods))))
            ((null methods))
            (and (eq selector (first methods))
                 (return-from send (apply (second methods) an-object
                                          selector (rest classes) args)) ) )))
;;; run-super  [Primitive 4]
(defmacro run-super (&rest args)
   (let ((classes (gensym))
         (methods (gensym))
         (args (gensym)) )
    '(block run-super
       (do ((,args (list ,@args))
            (,classes *classes* (rest ,classes)))
           ((null ,classes) (send an-object 'error "Method ~S not found" *selector* ,args ))
           (do ((,methods (iv-value (first ,classes) 'methods) (rest (rest ,methods))))
               ((null ,methods))
               (and (eq *selector* (first ,methods))
                    (return-from run-super (apply (second ,methods)
                                                  self *selector*
                                                  (rest ,classes) ,args))) )))))
;;; iv-value  [Primitive 5]
(defun iv-value (an-object name)
     (multiple-value-bind (description structure)
              (if (eq an-object (class-of an-object))
                  (values (class-all-iv an-object) (class-organisation an-object))
                  (values (iv-value (class-of an-object) 'iv)
                              (iv-value (class-of (class-of an-object)) 'organisation)) )
              (do ;; a basic description is: (name pos initial-value)
```

```
                ((desc description (rest desc))
                ;; Here is an assumption, structure hold (constructor,accessor,modifier).
                 (acces (elt structure 1)) )
                ((null desc)  (send an-object 'error
                                        "This instance variable is unknown: ~S" name)
                (and  (eq name (elt (first desc) 0))
                      (return-from iv-value  (funcall acces
                                                      an-object
                                                      (elt (first desc) 1) )) ) )))
;;; set-iv-value, this form also can be defined inside iv-value...  [Prmitive 6]
(defun set-iv-value (an-object name value)
     (multiple-value-bind (description structure)
              (if (eq an-object (class-of an-object))
                  (values (class-all-iv an-object) (class-organisation an-object))
                  (values (iv-value (class-of an-object) 'iv)
   (iv-value (class-of (class-of an-object)) 'organisation)) )
              (do ((desc description (rest desc))
                   (acces (elt structure 2)) )
                  ((null desc)  (send an-object 'error
                                        "This instance variable is unknown: ~S" name)
                  (and  (eq name (elt (first desc) 0))
                        (return-from set-iv-value
                             (funcall acces
                                      an-object
                                      (elt (first desc) 1) value) ) ) ) ) )
;;; setf form
(defsetf iv-value set-iv-value)
;;; basic class structure
;;; #( isit [(isit iv name supers linerar-extension organisation) .... ])
;;;  [Prmitive 7]
(defun class-all-iv (an-object) (elt (elt an-object 1) 1) )
;;;  [Prmitive 8]
(defun class-organisation (an-object) (elt (elt an-object 1) 5) )
;;; Class hand make...
 ...
;;; First bootstrap
  ...
;; First Object creation
(send class 'new
         :name 'object
         :supers ()
         :iv '(isit)
```

```
        :methods '(initialize
                    (lambda (args)
                         (mapc  #'(lambda (iv)
                                      ((lambda (find)
                                            (and find
                                                 (setf (iv-value self (elt  iv 0))
                                                       (second find))))
                                       (member (intern (symbol-name (elt iv 0)) 'keyword)
                                               args) ) )
                                  (iv-value (class-of self) 'iv))
                         (setf (iv-value self 'isit) (class-of self)) ) ) )
; Then we rewrite the class definition...
(send class 'new
        :name 'class
        :supers '(object)
        :iv '(iv name supers linear-extension organisation methods)
         :methods  '(basicNew (lambda ()
                                 (funcall (elt (iv-value self 'organisation) 0)
                                          self
                                         (iv-value self 'iv)) )
                     new (lambda (&rest args)
                                (send (send self 'basicNew) 'initialize args) )
                     initialize (lambda (args)
                                        ;; Feed slots
                                        (run-supers args)
                                        ;; compute linear extension.
                                        (send self 'compute-linear-extension)
                                        ;; collect all slots.
                                        (send self 'collect-slots)
                                        ;; walk-methods
                                        (send self 'code-walker)
                                        ;; here we are.
                                        (setf (symbol-value (iv-value self 'name)) self)
                                        self)
                     compute-linear-extension (lambda () ... )
                     collect-slots (lambda () ... )
                     error (lambda (&rest args) ... )
                     code-walker (lambda () ... ) ) )
;; Then complete the bootstrap by redefining Object and adjust all pointers.
```