Metaclasses are First Class : the ObjVlisp Model

Pierre Cointe
Rank Xerox & LITP

RXF: DRBI, 12 Place de l'Iris - Cedex 38, 92071 La défense
LITP: Université Paris-6, 4 place Jussieu, Tour 55-65, 75223 Paris
Email: ...!seismo!inria!litp!pc cointe@inria.inria.fr.uucp

Abstract

This paper shows how an attempt at a uniform and reflective definition resulted in an open-ended system supporting ObjVlisp, which we use to simulate object-oriented language extensions.

We propose to unify Smalltalk classes and their terminal instances. This unification allows us to treat a class as a "first class citizen", to give a circular definition of the first metaclass, to access to the metaclass level and finally to control the instantiation link. Because each object is an instance of another one and because a metaclass is a real class inheriting from another one, the metaclass links can be created indefinitely.

This uniformity allows us to define the class variables at the metalevel thus suppressing the Smalltalk-80 ambiguity between class variables and instance variables: in our model the instance variables of a class are the class variables of its instances.

1 The Instantiation Mechanism

1.1 Classes & Metaclasses

We focus on the instantiation mechanism of object-oriented languages which organizes objects in taxonomies along the class abstraction. Let us recall that the class concept invented by Simula and reimplemented by Smalltalk-72 is used to express the behavior of a set of objects which share the same semantics operating on the same attributes. This approach considers a class as a mould, manufacturing pieces called its instances. Alternatives to the class model - allowing other organizations of knowledge - are well known, for instance Hewitt's actor model. An actor describes its own structure and exists without a class. Defining generator actors, using a copy mechanism [9] or designing a delegation mechanism [15] are other themes developed by O.O programming.

"One way of stating the Smalltalk philosophy is to choose a small number of general principles and apply them uniformly" [14].

In most common class-oriented languages, despite Krasner's uniformity principle, a class is not a REAL object. Some of them however, like Loops [1], Smalltalk-80 [11], CommonLoops [2] and CLOS [3] introduced the metaclass concept to provide greater abstraction by allowing the description of a class by another class.

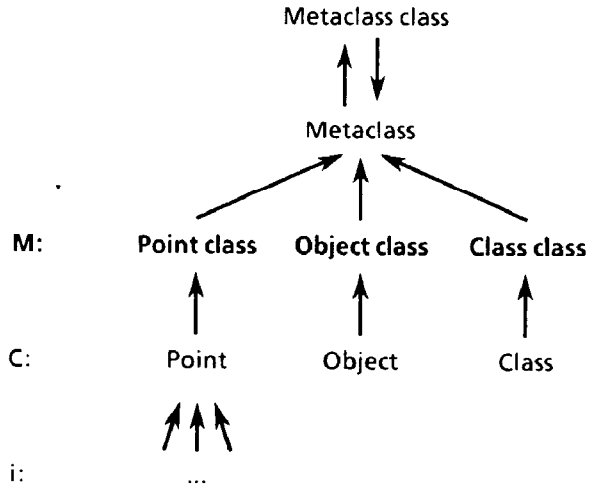
1.2 Smalltalk-80

"The primary role of a metaclass in the Smalltalk-80 system is to provide protocol for initializing class variables and for creating initialized instances of the metaclass's sole instance" (P. 287 [11]).

Smalltalk-80 uses the metaclass level facility to define the behavior of a class as the behavior of a regular object reacting to message passing (1). The role of a metaclass is to (re)define the instantiation method (3,4), to control the class variables initialization (2) or to explicitly explain the semantics of a class by predefined examples (5):

(1) Point class
(2) DeductibleHistory initialize
(3) a_point$_1$← Point new
(4) a_point$_2$← Point x: 20 y: 40
(5) Pen example

However, a Smalltalk metaclass is not a ordinary class but an anonymous one (accessible by the unary selector "class"), and which cannot be defined explicitly by users. This metaclass which supports the definition of the class instance variables and the class methods cannot exist without the class that is its only instance. Conversely, Smalltalk-80 associates a private metaclass to each class being created. Two classes cannot share the same metaclass. The next figure summarizes the Smalltalk-80 instantiation hierarchy :



Because every metaclass is automatically an instance of the `Metaclass` class, metaclasses are not true classes, the number of metalevels is fixed and the metalink cannot be created indefinitely [17].

From an inheritance point of view, the metaclass inheritance is also implicitly fixed by the system. The hierarchy of the metaclasses is parallel to the hierarchy of the classes. Because `Object` has no super class, the rule keeping a parallel hierarchy between classes and metaclasses, does not apply, and `Object class` is a subclass of the abstract class `Class`. Then, all Smalltalk metaclasses are subclasses of `Object class`, itself a subclass of `Class` :

```
Object ()
  Point (x y)
  Behavior (superclass methodDict format subclasses)
    ClassDescription (instanceVariables organization)
      Metaclass (thisClass)
      Class (name classPool sharedPools)
        <all metaclasses>
        Object class ()
          Point class ()
          Behavior class ()
          ...
```

Consequently, a metaclass cannot be defined ex nihilo as the subclass of a chosen class. This inheritance tree establishes the dichotomy between classes prototyped by the class `Class` and the metaclasses prototyped by the class `Metaclass` (even if they are both subclasses of `ClassDescription` and `Behavior`). These limitations, and the fact that a metaclass cannot exist without its class, introduce a first boundary between the implementor - who controls the metaclass level - and the user - who only has access to the class level. A second boundary is made apparent by the introduction of the `instance method` and `class method` terminologies (cf. the instance class SwitchView of the browser).

1.3 Loops

"For some special cases, the user may want to have more control over the creation of instances. For example, Loops itself uses different Lisp data types to represent classes and instances. The New message for classes is fielded by their metaclass, usually the object MetaClass. This section shows how to create a metaclass. Any metaclass should have Class has one of its super classes and MetaClass as its metaclass. The easiest way to create a new metaclass is to send a New message to MetaClass as follows : (← ($ Metaclass) New metaCLassName '(Class))" (P. 86 [1]).

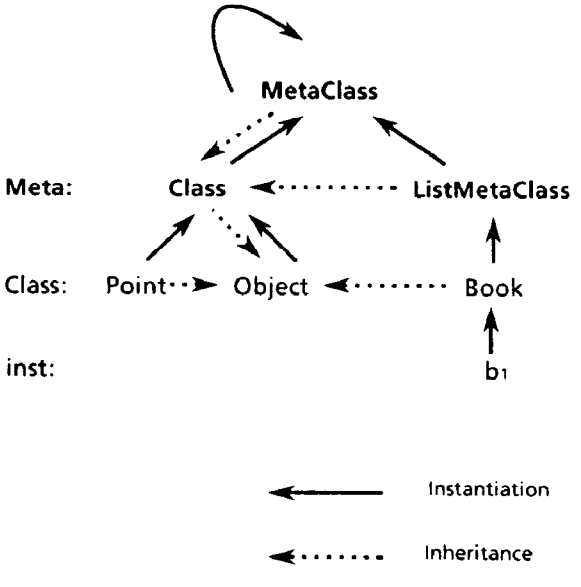
The Loops scheme for metalevels is close to the Smalltalk scheme. The basic idea is to introduce three levels corresponding to three kinds of object: instances, classes and metaclasses. This scheme is built on the "Golden Braid" `Object`, `Class` and `MetaClass` (cf. P. 113 [1]);

- `MetaClass` is the class which holds the default behavior for metaclasses as objects, it is the metaclass of all other metaclasses and its own metaclass. `MetaClass` holds the `New` method which creates class data-types.
- `Class` is the class which holds the default behavior for classes as objects. `Class` is the default metaclass for all classes. It holds the `New` method which creates instance data-types. Consequently, if `Class` is not the metaclass for a class, it must be on the supers list of that metaclass (which inherits its new method). According to this rule `Class` is a super of `Metaclass`.
- `Object` is the class which holds the default behavior for all instances. Consequently, `Object` is the root of the inheritance tree.

These three classes are used to create new metaclasses (1), new classes (2 3) and new instances (4 5) :

```
(1) (← ($ Metaclass) New 'ListMetaClass '(Class))
(2) (← ($ Class) New 'Point '(Object))
(3) (← ($ ListMetaClass) New 'Book)
(4) (← ($ Point) New 'a_point1)
(5) (← ($ Book) New 'b1)
```

The next figure summarizes the instantiation and subclass links provided by Loops (the black arrow means `instanceOf`, the shaded arrow means `subclassOf`) :



Unlike Smalltalk-80, a metaclass can be created explicitly as the subclass of another one but must be an instance of `MetaClass`. This last condition fixes the depth of the instantiation tree and leads the Loops implementors to use a non uniform representation for classes and terminal objects. On the other hand, the Loops manual does not express any circular definition of `MetaClass` as it would be suggested by its self-instantiation.

1.4 Unification

To suppress the gap between class and object, we propose a unification of the metaclass, class and object concepts. We claim that a class must be an object defined by a real class allowing greater clarity and expressive power.

The reverse question is "Is every object a class? ". The answer is no : some objects are only instances of a class and do not define a model. An instance of a Point class, e.g. an object $\texttt{a\_point}_1$, or an instance of the Number class, e.g. 3, are such non-instantiable objects. We call them terminal instances.

Thus we consider only one kind of object, without distinctions of structure or type between classes and terminal instances (non-class). 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 class. Otherwise it is a terminal instance. A metaclass is simply a class which instantiates 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 discrimination between metaclasses, classes and terminal instances[1] is only a consequence of inheritance and not a type distinction. We can distinguish between class and non-class objects, however the ObjVlisp model takes into account only one type of object.

This unification simplifies the instantiation and inheritance concepts, using them simultaneously : for example, a metaclass must be 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 [8]. Our wish is to present a synthesis, using operational semantics expressed in Lisp. We present here the reflective version which integrates the previous unification and gives a good solution to the problem of the *<class , instance>* dichotomy.

[1]To easily distinguish them, we use upper-case initial letters for classes and metaclasses plus bold letters for metaclasses, and lower-case letters for terminal instances.

2.1 ObjVlisp in six Postulates

Following the classical presentation of Smalltalk-76 [13], six postulates fully describe the ObjVlisp model :

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

$$\boxed{\text{object} = < \text{data} , \text{procedures} >}$$

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

$$\boxed{(\text{send object } \texttt{selector } \text{Args}_2 \ldots \text{Args}_n)}$$

P3: every object belongs to a class that specifies its data (attributes called fields) and its behavior (procedures called methods). Objects will be dynamically generated from this model, they are called instances of the class. 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, instantiated by another class, called its metaclass. Consequently (P3), to each class is associated a metaclass which describes its behavior as an object. The initial primitive metaclass is the class **Class**, built as its own instance.

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 behavior 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 according to the following equation :

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

2.2 Classes and objects

Structure of an object

The postulates P1 & P3 & P6 define an object as a "chunk" of knowledge and actions whose structure is defined by its class. More precisely:

Fields : are the set of variables defining the environment of the object;

a) **instance variables :** this first environment is organized as a "dictionary" split into two isomorphic parts :

1. the set of instance variables specified by the object's class,
2. the set of associated values.

The set of instance variables belongs to the class, and is shared by all its instances. The set of values is owned by each instance; consequently an object cannot exist without its class. These two sets [2] are ordered - at creation time - by the inheritance rules defined on the superclasses components. In particular, to store the name of its class, each object holds as first instance variable one named `isit`. Each object holds also the `self` "pseudo instance variable" dynamically (at runtime) bound to the object itself when it receives a message. These two pseudo variables are respectively analogs to `isit` and `self` in Smalltalk-72 [10].

b) **class variables** : Smalltalk-80 class variables are accessible to both the class (via instance methods) and its metaclass (via class methods). Similarly, ObjVlisp instance variables of a class (defined as an object) are accessible to both its own methods and its metaclass methods. Consequently, the instance variables of a class are also the class variables of its instances, defining global environment at the metaclass level.

Methods : The methods define the procedures shared by all the instances of a class and owned by the class. To realize the unification between class and instance, we represent the method environment as a particular instance variable of its metaclass; the methods dictionary of a class is the value associated with a specific instance variable called methods. As a common object, a class is defined by its class and the values of the associated instance variables.

Structure of a class

As an object, a class owns also the `isit` instance variable inherited from Object (cf. 3.2). This variable is bound to the name of the metaclass when the class is created. Because a class is also a generator of objects , we have to introduce the minimal set of instance variables describing a class. Four explicit instance variables are owned by **Class** as the primitive metaclass:

1. `name` : the name of the class, which means that each class is a non-anonymous object,
2. `supers` : the list of the direct superclasses of the class,
3. `i_v` : the list of instance variables that the class specifies,
4. `methods` : the method-dictionary e.g. the list of methods held by the class expressed as a "P-list", with pairs <selector , λ-expression>.

[2] Each object is implemented as a pointer to an abstract structure (for example a list, a vector, a hash-table or a Lisp structure) which must be isomorphic to the list of instance variables held by its class :

object = #(class-name i_v_2* ... i_v_n*)
i_v [class-name] instantiate #(class-name i_v_2* ... i_v_n*)

Instantiation of a class

Unlike the Smalltalk-80 and Loops systems, ObjVlisp uses only one method to create an object which can be a terminal instance or a class.

Basicnew : this method is owned by the metaclass **Class** and uses the `makeInstance` primitive of the virtual machine [3] as expressed by the circular definition of **Class** (cf 3.1). This method implements only the allocation of the structure with the `nil` default-value for each instance variable :

(send Aclass 'basicnew) ⇒ #(Aclass nil ... nil)

New : to allocate a new object and to initialize its instance variables, ObjVlisp uses the `new` method, owned by **Class**. This `new` method has two effects: to allocate a new object and to give an initial value to each instance variable. To distinguish these two functions, `new` composes the `basicnew` method with one of the two `initialize` methods defined respectively in **Class** and **Object**.

Consequently, the instantiation semantic and syntax are totally uniform: the `new` message sent to a class always receives as arguments the values related to the instance variable specified by the receiver class and creates a new instance built on the class model. To allow more expressive power each argument of the `new` message must be prefixed by a keyword (for example :name for an instance variable called *name*) denoting the instance variable receiving the associated value:

(send Aclass 'new :i_v_2 i_v_2 ... :i_v_n i_v_n) ⇒
#(Aclass i_v_2* ... i_v_n*)

Examples : we define the class Point by instantiating the metaclass **Class**; the receiver (here **Class**) specifies the name of the model (the value of the implicit `isit` instance variable) and the values associated to the four instance variables of **Class** must be expressed :

```
(send Class 'new
  :name       'Point
  :supers     '(Object)
  :i_v        '(x y)
  :methods    '( x        (λ () x)
                 x:       (λ (nv) (setq x nv) self)
                 init     (λ () (setq x 40 y 12) self)
                 display  (λ ()
                            (format ()
                              (catenate "~" x "D")
                              "*") ) ) )
```

Then we create instances of Point, using the same `new` message[4] :

```
(setq a_point1 (send Point 'new :x 20 :y 30))
(setq a_point2 (send Point 'new :y 30 :x 20))
(setq a_point3 (send Point 'new))
(setq a_point4 (send Point 'new :x nil :y nil))
(setq a_point5 (send Point 'basicnew))
```

[3] The `makeInstance` function creates a new object when receiving as argument the name of its class :

```
(defun makeInstance (aclass)
  (tcons aclass (makelist (1- (length (send aclass 'i_v))) nil)))
```

The auto-quoted keywords suppress the order of instance variables values e.g. the two objects a_point_1 and a_point_2 are equal. Keywords may be also omitted, in this case the associated instance variable is bound to the `nil` default value (e.g. a_point_3, a_point_4 and a_point_5 are equal).

Obviously, the user can modify the behaviour of the `new` message by defining the `initialize` method at a subclass level. For instance, to create all the instances of Point with the `0` value for `x` and `y`, we will redefine in Point the `initialize` method : `(λ (i_v*) (setq x 0 y 0) self)`

3 From Uniformity to Reflection

Since giving complete control to the users means a complete transparency in the objects definitions, we adapt the reflective interpreter technique [16] to the construction of this model. ObjVlisp is supported by two graphs: the instantiation graph and the inheritance graph. The instantiation graph represents the `instanceOf` relationship (P3 & P4), and the inheritance graph the `subclassOf` link (P5). Class and Object are the respective roots of these two (acyclic) graphs: they are defined in ObjVlisp as follows.

3.1 Class: Instantiation

Class is the first object of the system. As the root of the instantiation graph, it defines the behavior for classes. Because the new primitive is fielded by Class it will recursively create all other objects. To prevent the infinite regress provided by the instantation link (a metaclass is a class which instantiates a class, a metametaclass is a class which instantiates a metaclass, a metametametaclass ...), Class must be its own instance which severely constrains its structure.

Reflective pattern matching of Class

To verify the previous statement, we have to guarantee that the instance variables specified by Class match the corresponding values also held by Class, as its own instance, which is easily obtained by :

isit	name	supers	i_v	methods
Class	Class	(Object)	(isit name supers i_v methods)	(new (λ..)..)

Notice that the value associated with the instance variable `i_v` is exactly the ordered set of instance variables `(isit, name, ... , methods)` itself, this reflective pattern matching illustrates the definition of Class as an object.

[4] This table shows the dictionary of values owned by each object :
? (send a_point_1 'i_values)
= #(Point 20 30)
? (send Point 'i_values)
= #(Class Point (Object) (isit x y) (x λ_1 x: λ_2 init λ_3 display λ_4))
? (send Class 'i_values)
= #(Class Class (Object) (isit name supers i_v methods) (new...))

To prepare the bootstrap: the Lisp skeleton

"A natural and fundamental question to ask, on learning of these incredibly interlocking pieces of software and hardware is: "How did they ever get started in the first place?". It is truly a baffling thing" [12].

Defining Class from itself necessitates specifying the bootstrap mechanism. We create manualy the skeleton of Class. If we represent objects as lists, we will use the skeleton :

```
(setq Class '(
  Class
  Class
  (Object)
  (isit name supers i_v methods)
  (
   new          (λ (self . i_values)
                  (send (makeInstance name)
                        'initialize i_values)
   initialize   (λ (self i_values)
                  (initIv self ... ...)
                  (setq i_v (herit-i-v supers i_v))
                  (setq methods (scan-methods ...))
                  (set name self)) ))
```

In fact, we only define the new and `initialize` methods supporting the self-instantiation of Class.

This bootstrapping process then creates the real Class object, by sending to the Class skeleton the appropriate new message. Note that the skeleton is destroyed by the circular (re)definition of Class.

The bootstrap: the Self Instantiation of Class

The Class definition establishes that Class is its own instance, is a subclass of Object, and uses the instance variables previously mentioned. These definitions and examples are given for Le_Lisp [7] :

```
(send Class 'new
  :name      'Class
  :supers    '(Object)
  :i_v       '(name supersi_v methods)
  :methods   '(
               new          (λ i_values
                              (send (send self 'basicnew)
                                    'initialize i_values))
               basicnew     (λ () (makeInstance name))
               initialize   (λ (i_values)
                              (run-super)
                              (setq i_v (herit-i-v supers i_v))
                              (setq methods ...))
                              (set name self))
                 ...                ...

               name         (λ () name)
               supers       (λ () supers)
               i_v          (λ () i_v)
               methodsDic   (λ () methods)
                 ...                ...
               understand   (λ (selector method)
                              (defmethod self selector method))
               selectors    (λ () (selectors methods)) ) )
```

The definitions of the methods shows that all instance variables are automatically bound to their values in a method body. Consequently, the λ-expressions associated with the `name`, `supers`, `i_v` and `methods` selectors are quite easy to express. Similarly, in the `new` method, `self` denotes the generator (here Class). The `initialize` method uses the `run-super` form to call the general allocator (`makeInstance`) defined at the Object level.

3.2 Object: Inheritance

Postulate (P5) introduces the inheritance mechanism (which concerns only classes). The ObjVlisp inheritance allows to connect together instance variables and methods of several classes but in two different ways:

- The inheritance of instance variables 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 superclasses with the instance variables specified at creation (the value associated to the ":i_v" keyword used by the new message).

- On the other hand, method inheritance is dynamic and uses the virtual copy mechanism implemented by the linkage of classes in the inheritance graph which is supported by the `supers` instance variable. When the method lookup fails in the receiver class then the search continues in a depth-first/breadth-first way.

  This lookup call may be locally modified by the `run-super` form - same as in CommonLoops [2] and similar to the `super` construct of Smalltalk-80 - which overrides the current method. The lookup starts (statically) at the superclass(es) of the class containing the method.

Classes vs Terminal Instances: the initialize method

The inheritance mechanism of instance variables is applied only when creating classes. Thus we need to distinguish creation of classes and creation of terminal instances. As we pointed out already, a metaclass is a class which inherits from Class the `new` method and the (`name supers i_v methods`) instance variables[5].

The reflective definition of ObjVlisp allows to use only one allocator - the `basicnew` - and nevertheless to explicit the difference between class and terminal instance creations: the `initialize` method owned by Object treats the terminal instance, and the `initialize` method owned by Class implements the inheritance mechanism associated to instance variables at a class creation time.

Object the most common class

The second primitive class is Object, instance of Class. Object represents the most common class - the intersection of all classes - describing the most common behavior (for classes and terminal instances). It is created during the bootstrap mechanism, immediately before Class. The `isit` instance variable is statically inherited by all classes. Then `isit` provides the instanciation link (the umbilical cord) between a class and its instances.

```
(send Class 'new
  :name       'Object
  :supers     '()
  :i_v        '(isit)
  :methods    '(
                class        (λ () isit)
                initialize   (λ (i_values)
                               (initIv self ... ...) self)
                ?            (λ (i_var) (ref i_var self))
                ?←           (λ (i_var i_val)
                               (setf (ref i_var self) i_val))
                i_values     (λ () self)
                metaclass?   (λ () (memq 'supers i_v))
                class?       (λ ()
                               (send (i_v* isit) 'metaclass?))
                  ...          ...
                error        (λ msg '(λ bs ',msg)) ) )
```

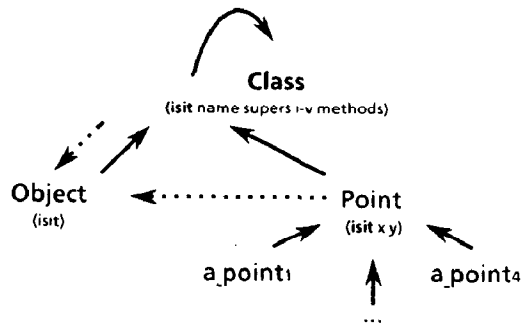
From this definition Object has no superclasses and each ObjVlisp object answers to the <*selector*> by <action>:

class	giving the name of its class
initialize	initializing the instance variables
?	returning the value of the field i_var
?←	writing i_var with the new value
i_values	returning the list of values of the i_v
metaclass?	testing if the object is a metaclass
class?	testing if the object is a class
error	implementing the standard treatment of error

Notice that the ? and ?← methods which access the value of any instance variable (read&write) respect their lexical scoping (and violate the encapsulation principle).

3.3 Architecture of the ObjVlisp model

We summarize the general structure of the ObjVlisp model by connecting together the `instantiation graph` and the `inheritance graph`. At the creation of the system there are only the Class and Object classes. The "naive" use of the system will keep the depth of the instantiation tree to three. See below for a similar example to Smalltalk-76 [13]; all classes are instances of Class :



[5] The *metaclass?* predicate defined in Object uses the `supers` instance variable to recognize metaclasses.

The rest of this paper establishes that creation of metaclasses brings substantial benefits. There is no longer any depth limitation of the instantiation tree, and the user can extend it as much as he wants to specify different metalevels of shared instance variables and methods.

4 From Reflection to Extensibility

4.1 Building new metaclasses

By combining the inheritance mechanism with the instantiation one we can create multiple metaclasses. **Ametaclass** is defined as a subclass of **Class** i.e. dynamically inherits the **new** primitive (to create objects) and it receives a copy of the basic instance variables defining a class (`name`, `i_v`, `supers` and `methods`), copy extended by the cv_i variables :

(send Class 'new	
:name	'Ametaclass
:i_v	'(cv_1 ... cv_n)
:supers	'(Class)
:methods	'(...))

Following this definition, the creation of Aclass needs the instantiation of every basic instance variables plus the instantiation of each new cv_i :

(send Ametaclass 'new	
:name	'Aclass
:i_v	'(iv_1 ... iv_n)
:supers	'(...)
:methods	'(...)
:cv_1	cv_1*
...	...
:cv_n	cv_n*)

Class variables by Example

Let us return to the Point class, previously defined. Now we would like the constant character * 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 us call it **MetaPoint**) specifies this common character :

(send Class 'new		
:name	'MetaPoint	
:supers	'(Class)	
:i_v	'(char)	
:methods	'())	
(send MetaPoint 'new		
:name	'DefaultPoint	
:supers	'(Object)	
:i_v	'(x y)	
:methods	'(*init*	(λ () (setq x 40 y 12) self)
	x	(λ () x)
	x:	(λ (nx) (setq x nx) self)
	display	(λ () (format () (catenate "~" x "D") char)))
:char	"*")	

MetaPoint is declared as a subclass of **Class** (thus it is a metaclass). It inherits the `name`, `supers`, `i_v` and `methods` instance variables from **Class** and adds to them the instance variable `char`. Consequently, DefaultPoint specifies the associated value of `char`, i.e. * by using the associated keyword. Now we could create such a point :

```
? (setq a_point (send DefaultPoint 'new :x 20))
= a DefaultPoint
? (send a_point 'display)
=          *
```

Class methods by Example

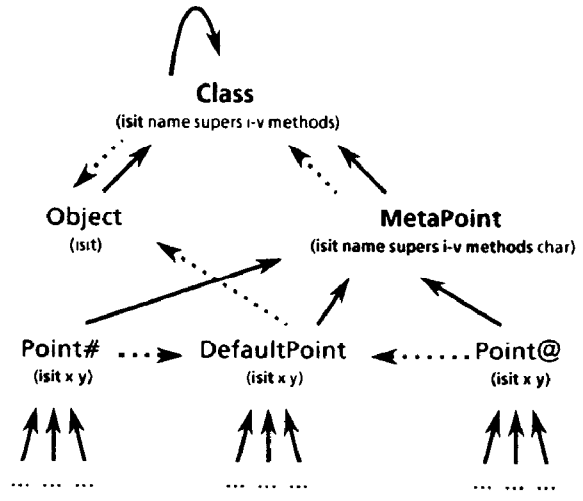
As for class variables, class methods are specified in the metaclass as ordinary methods. Suppose we want to define a new class method for DefaultPoint to create and initialize a new point. We simply define the `newinit` method of `MetaPoint` (assuming we define also an `init` method in the Point class, or at least in the Object class) :

(send Class 'new		
:name	'MetaPoint	
:supers	'(Class)	
:i_v	'(char)	
:methods	'(*newinit*	(λ () (send (send self 'new) 'init)))
	char	(λ () char)
	char:	(λ (newChar) (setq char newChar))))

- `newinit` creates a new instance, `(send self 'new)` then receives the `init` message.
- `char` gives access to the char variable. We have introduced this method to show that `char` is both accessible by the DefaultPoint `display` method and by the MetaPoint `char` method,
- `char:` allows the modification of the `char` class variable. For instance, the `(send DefaultPoint char: ''@'')` message provides the new @ display for all the instances of DefaultPoint.

4.2 Parametrization of a class

The DefaultPoint class is now parametrized by its display character and the **MetaPoint** metaclass represents this abstraction. Let us define two new classes, called Point# and Point@ with two different display characters. Obviously, they are defined as a subclass of DefaultPoint :



Notice that a same metaclass (here **MetaPoint**) can be used to instantiate several classes (Point#, Point@ ...) : there is no one specific metaclass associated to each class.

```
? (send MetaPoint 'new
        :name 'Point# :supers '(DefaultPoint) :char "#" )
= Point#
? (send 'MetaPoint 'new
        :name 'Point@ :supers '(DefaultPoint) :char "@" )
= Point@
? (send (send Point# 'new :x 1) 'display)
= #
? (send (send Point@ 'new :x 9) 'display)
=       @
```

Comparison with Smalltalk-80

We have pointed in [5] that the Smalltalk-80 terminology is not homogeneous. Smalltalk class variables are not the instance variables of the class defined as an object but a dictionary of variables shared between all the instances of a same class hierarchy. For example, if the new method is redefined to add the newly created instance of a class inside a Collection's class variable, the instances of its subclasses will also be memorized.

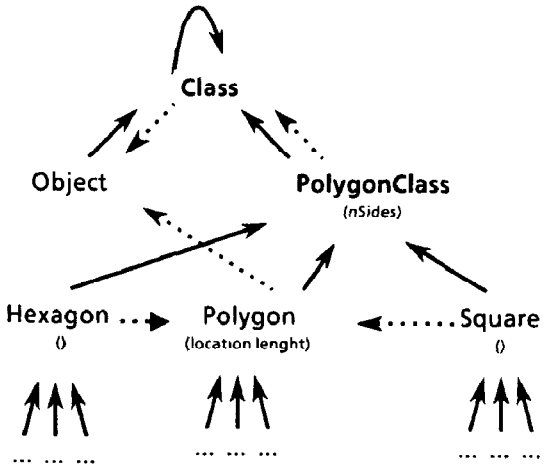
Nevertheless, if we use the instance variables of a metaclass, we can simulate the "MetaPoint" construction in Smalltalk-80. Obviously we need to give an explicit access to the char variable and we have to use a new different metaclass for each class of Point :

```
Object subclass: #DefaultPoint
  instanceVariableNames: 'x y'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Graphic-Primitives'.
DefaultPoint class instanceVariableNames: 'char'.
DefaultPoint class methodsFor: 'meta-iv access'
  initialize ↑char← "".

DefaultPoint subclass: #Point@
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Graphic-Primitives'.
Point@ class instanceVariableNames: ''.
Point@ class methodsFor: 'meta-iv access'
  initialize ↑char← "@".
```

The DefaultPoint example illustrates a general knowledge scheme (as does the Polygon example below). To generalize the solution, we have decided to extend the scope of the instance variables of a class to each of its instances. Unlike Smalltalk-80, our class variables are inherited but not shared by the subclasses.

Each polygon is defined by its `location` (the first vertex) and the `length` of any of its sides. To parametrize the number of sides - 4 for a square, 6 for a hexagon, undef for a polygon - we use the `nSides` class variable. To simplify the next figure, the inherited variables are not drawn :



4.3 Filiation link (Set) :

To use classes which remember all their instances, we define a new metaclass (**Set**), as a subclass of **Class** with the new **sons** instance variable pointing the list of instances. We just have to redefine the **new** method in **Set** to add the newly created instance at the end of the sons list :

```
(send Class 'new
  :name        'Set
  :supers      '(Class)
  :i_v         '(sons)
  :methods     '( sons      (λ () (cdr sons))
                  new       (λ i_values
                              (nconc sons
                                  (cons (run-super) ())))
                  mapsons   (λ (unaryS)
                              (mapc
                                (λ (rec) (send rec unaryS))
                                (send self 'sons))
                              unaryS) ) )

(send Set 'new
  :name        'Point
  :i_v         '(x y)
  :supers      '(Object)
  :methods     '( init      (λ () (setq x 40 y 12) self)
                  display   (λ () (print (format () ...)) self) )
  :sons        '(hook) )
```

The **sons** method gives access to the sons class variable and the **mapsons** method distributes an unary message (without arguments) to all the instances of a particular set. The next session shows the behavior of Point defined as an instance of **Set**

```
? (prog1 'ok (send Point 'new :x 10) (send Point 'new :x 20))
= ok
? (send Point 'sons)
= (#(Point 10 nil) #(Point 20 nil))
? (send Point 'mapsons 'display)
      *
            *
= display
```

This solution provides a uniform extension of the metaclass system and seems better than the Loops "class property hook" used by the **ListMetaClass** definition in [1] (P. 36).

4.4 MetaPoint as an instance of Set :

Now we can add a new metaclass level by defining **MetaPoint** as a subclass of **Set**, allowing DefaultPoint, Point# and Point@ to memorize their instances. Since a metaclass is a ordinary class, such an extension is easy to repeat and the metalinks can be created indefinitely.

5 Metaclasses are useful

"With respect to Simula, Smalltalk also abandons static scoping, to gain flexibility in interactive use, and strong typing, allowing it to implement system introspection and to introduce the notion of meta-classes [6]."

5.1 Metaclasses provide metatools to build open-ended architecture

"The metaclass determines the form of inheritance used by its classes and the representation of the instance of its classes. The metaclass mechanism can be used to provide particular forms of optimization or to tailor the Common Lisp Object System for particular uses (such as the implementation of other languages like Flavors, Smalltalk-80 and Loops)" [3].

From an implementor's point of view, metaclasses are very powerful because they provide hooks to extend or modify an existing kernel. For example, ObjVlisp uses the metaclass facilities to simulate other object-oriented systems. Metaclasses may control :

1. the inheritance strategy (simple, multiple, method wrapping [18]). To implement variations on inheritance schemes [19], we define at the metaclass level a method (or an instance variable) parametrizing the `lookup` method used by the `send` primitive,

2. the internal representation of objects by using different `makeInstance` primitives creating lists, vectors, hashtables or structures; each metaclass fielding a private `new` method,

3. the access to methods by implementing a caching technique. We associate with each class a private memory (the `cache` instance variable) memorizing the addresses of methods already called,

4. the access to instance variable values by distinguishing between private and public variables or by implementing active-values or demons.

5.2 Metaclasses remove the boundary between users and implementors

In our empirical studies, metaclasses were regarded as the most significant barrier to learnability by both students and teachers. We propose that they be eliminated. We have explored various alternatives to metaclasses, such as the use of prototypes. However, for DeltaTalk we simply propose that the language revert to the situation in Smalltalk-76. Every class would be instance of class Class" [4].

Obviously we disagree with the Borning's conclusion. We consider that metaclasses provide an explicit definition of the class system. They express the behavior of classes in a transparent way. Because they have ability to manipulate their own structures, they can implement system introspection. Consequently, metaclasses support a circular definition of the system reducing the boundary between users and implementors. But, to fully exploit this metalevel, the metaclass concept must be simple enough to be understood by the user. We believe this is not true in Smalltalk-80 but that the ObjVlisp uniform and reflective architecture has reached this goal.

6 Conclusions

6.1 Results

The ObjVlisp model's primary advantage is **uniformity**. There is now only one kind of object: a class is an object and a metaclass is a true class whose instances are classes. This allows a simplification and economy of concepts, which are thus more powerful and general. The second property is **reflection** which provides a language completely and uniformly accessible by the user. The system is self-described by the explicit definition of the root of the instantiation tree (**Class**) and the root of the inheritance tree (Object). The main results are that there is no limitation in the depth of the instantiation tree, the metalinks can be created indefinitely and class variables are defined at the metaclass level. Finally, **extensibility** permits various applications and modeling alternative semantics, for instance Thinglab composite objects and partwhole hierarchy or Smalltalk-80 dependencies [11].

6.2 New Improvements

A first version of this paper was presented at the *workshop on Meta-Level Architectures and Reflection* organized in Alghero [16]. In this new version, the difference between instance creation and class creation is explicitly defined at the ObjVlisp level through two distincts `initialize` methods, respectively owned by Object and `Class`. Thus we do not need to add an extra metalevel (the metaclass level of Loops or Smalltalk-80) [5] and the ObjVlisp instantiation kernel is really minimal.

6.3 Future work

We have used the ObjVlisp model to study the instantiation mechanism. We plan now to investigate three axes:

- experimentation in object-oriented methodologies by writing relevant examples in ObjVlisp other than those provided by the Smalltalk-80 image,

- development of an object kernel for EuLisp and IsoLisp. This work is very close to the CLOS approach [3] but we expect to use the ObjVlisp experience to propose a cleaner metaclass level,

- implementing the ObjVlisp metaclasses architecture in Smalltalk-80 by redefinition of the "kernel classes".

Acknowledgements

We thank Jean-Pierre Briot for its major contribution to the ObjVlisp model, Alain Deustch for its implementation of the tree-walker, Jean-François Perrot, Henry Lieberman, Kris Van Marcke, Glenn Krasner and Nicolas Graube for their helpful comments on this text.

The ObjVlisp project is part of the "O.O.P. Methodology" group of the GRECO de Programmation

References

[1] Bobrow, D.G., Stefik, M., The LOOPS Manual, Xerox PARC, Palo Alto CA, USA, December 1983.

[2] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F., CommonLoops: Merging Lisp and Object-Oriented Programming, *OOPSLA'86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 17-29, Portland OR, USA, November 1986.

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

[4] Borning A., O'Shea, T., DeltaTalk: An Empirically and Aesthetical Motivated Simplification of the Smalltalk-80 Language, *ECOOP'87, to appear in Springer Verlag, Bezivin J. & Cointe P. ed.*, Paris, France, 15-17 June 1987.

[5] Briot, J-P., Cointe, P., A Uniform Model for Object-Oriented Languages Using the Class Abstraction, *IJCAI'87*, Milan, I, August 1987.

[6] Cardelli, L., A Semantics of Multiple Inheritance, *Bell Laboratories*, Murray Hill NJ, USA, 1984.

[7] Chailloux, J., Devin, M., Dupont, F., Hullot, J.M., Serpette, B., Vuillemin, J., LE LISP de l'INRIA, Version 15.2 (The manual), INRIA, Domaine de Voluceau, Rocquencourt 78153 le Chesnay, Mai 1986.

[8] Cointe, P., A VLISP Implementation of SMALLTALK-76, pp 89-102, *Integrated Interactive Computing Systems*, North-Holland, Degano, P. & Sandewall, E. editors, 1983.

[9] Cointe, P., Briot J.P., Serpette B., The FORMES language: a Musical Application of Object Oriented Concurrent Programming, pp 221-258 in Object Oriented Concurrent Programming, MIT Press, Cambridge, Mass A Yonezawa & M. Tokoro editors, May 1987.

[10] Goldberg, A., Kay, A., Smalltalk-72 Instruction Manual, *Research Report SSL 76-6*, Xerox PARC, Palo Alto CA, USA, March 1976.

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

[12] Hofstadter D.R., *GOEDEL, ESCHER, BACH: an Eternal Golden Braid*, The Harvester Press, John Spiers editor, Stanford Terrace, Hassocks, Sussex Publisher, 1979.

[13] Ingalls, D.H., The Smalltalk-76 Programming System Design and Implementation, *5th ACM Symposium on POPL*, pp. 9-15, Tucson AZ, USA, January 1978.

[14] Krasner, G., Smalltalk-80 - Bits of History - Words of Advice, Addison-Wesley, Reading MA, USA, 1983.

[15] Lieberman, H., Delegation and Inheritance, Two Modular mechanisms, Conf. Record of the 3rd Workshop on OOP, Centre Georges Pompidou, Paris, Bigre+Globule No 48, Bezivin J. and Cointe P. editors, January 1986.

[16] Maes, P., and al, Workshop on Meta-Level Architectures and Reflection, *to appear in North Holland, P. Maes & D. Nardi ed.*, Alghero, Italy, 27-30 October 1986.

[17] Maes, P., Computational Reflection, PhD thesis, Vrije Universiteit Brussel, AI-LAB Pleinlaan 2, B-1050 Brussels, Belgium, Mars 1987.

[18] Moon, D., Object-Oriented Programming with Flavors, *OOPSLA'86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 1-16, Portland OR, USA, November 1986.

[19] Stefik, M., Bobrow, D.G., Object-Oriented Programming: Themes and Variations, The AI magazine, pp 40-62, Winter 1985.

Appendix

We give two alternatives implementations written in Le Lisp representing objects as lists. The first one dynamically binds the instance variables at the run-time (cf. the `send` form), when the second one pre-compiles the methods by using a tree-walker. The `smacrolet` form replaces each instance variable name by its access function (cf. the `ref` form). For instance, below are the definition of `x`, `x:` and `display` methods of Point after their textual expansion:

```
(λ () x)                          (λ () (ref 'x self))
(λ ()                             (λ ()
  (format () ... char))             (format () ... (ref 'char (class-of self))))
(λ (nx)                           (λ (nx)
  (setq x nx) self)                 (setf (ref 'x self) nx) self)
```

```
; Dynamic access to instance variables

(typecn #/: (typecn #/a))
(defsharp |:| ()
   (with ( (typecn #/: 'cpkgc)  (typecn #/# 'cpname) )
      (list (read)) ))
(defvar class 'noboot) (defvar object ()) (defvar self ())

(dmd type-of (name) '(car ,name))
(dmd class-of (obj) '(i-v* (type-of ,obj)))
(dmd metaclass-of (obj) '(i-v* (metatype-of ,obj)))
(dmd metatype-of (obj) '(type-of (class-of ,obj)))
(dmd name (name) '(cadr ,name))
(dmd supers (name) '(caddr ,name)))
(dmd i-v (name) '(cadddr ,name))
(dmd i-v* (x) '(symeval ,x))
(dmd keywords (name) '(cadddr (cdr ,name))))
(dmd methodDic (name) '(cadddr (cddr ,name))))
(dmd methods (name) '(cdr (plist-to-dico ,name)))
(dmd selectors (name) '(car (plist-to-dico ,name)))
(dmd attach (s l) '(rplac ,l ,s (cons (car ,l) (cdr ,l))))
(dmd methodfor (classe sel) '(getl (methodDic ,classe) ,sel))
(dmd rewrite (inst isit) '(cval-dico (i-v (i-v* ,isit)) ,inst))

(defun send (obj -selector- . -args-)
  (if (eq obj self) (apply (lookup -selector- (i-v* isit) obj) obj -args-)
      (letv (i-v (metaclass-of obj)) (class-of obj)
         (letv (i-v (class-of obj)) obj
             (protect
                (apply (lookup -selector- (i-v* isit) obj) obj -args-)
                (rewrite obj isit))))))
(dmd run-super ()
  '(apply ,(lookup -selector- (i-v* (car (supers (i-v* isit)))) self) self -args-))

(defun makeInstance (model)
    (tcons model (makelist (1- (length (i-v (i-v* model)))) nil)))
(defun initIv (structure slots slots*)
  (while slots (set (nextl slots) (nextl slots*)))
  structure)

(defun getl (l sel)
   (cond
      ((not (consp l)) ())
      ((eq (car l) sel) (cadr l))
      (t (getl (cddr l) sel)) ))
(defun plist-to-dico (sexp)
   (when (symbolp sexp) (setq sexp (plist sexp)))
   (let ((sel (ncons nil)) (meth (ncons nil)))
    (letn self ((qsel sel) (qmeth meth) (l sexp))
      (ifn l (cons (cdr sel) (cdr meth))
           (self (placdl qsel (nextl l)) (placdl qmeth (nextl l)) l)))))
(defun cval-dico (dic-var dic-val)
   (when dic-val
     (rplaca dic-val (i-v* (car dic-var)))
     (cval-dico (cdr dic-var) (cdr dic-val))))))
```

```
(defun lookup (sel isit obj)
   (tag backtrack (tag again (depth sel isit)) (send obj 'error 'lookup)))
(defun depth (key node)
  (let ((method (methodfor node key)))
    (cond
      (method (exit backtrack method))
      ((eq node object) (exit again 'backtrack))
      (t (breadth key (supers node))) )))
(defun breadth (key l-nodes)
  (while l-nodes  (tag again (depth key (i-v* (nextl l-nodes))))))

(defun scan-match (pat dat-plist)
   (mapcar (lambda (msg) (getl dat-plist msg)) pat))
(defun make-keywords (iv)
   (mapcar (lambda (k) (setq k (concat ': k)) (set k k)) iv))
(defun scan-selectors (dic-methods)
   (mapc (lambda (sel) (make-descriptor sel descriptor)) (selectors dic-methods)))
(defun scan-iv* (f v)
   (let ((lv v) (p ()))
     (while f (if v (nextl v) (newl p ()))
     (nextl f))
   (nconc lv p))))
(defun scan-method (lambda-form classe)
   (scan-parameters lambda-form (cadr lambda-form)))
(defun scan-parameters (l lpar)
   (cond
     ((null lpar) (rplaca (cdr l) '(self)))
     ((atomp lpar) (rplaca (cdr l) (cons 'self lpar)))
     (t (attach 'self lpar))))
(defun scan-methods (pl classe)
   (ifn (cddr pl) (when (cadr pl) (scan-method (cadr pl) classe))
        (scan-method (cadr pl) classe)
        (scan-methods (cddr pl) classe))
    pl)
(defun defmethod (class sel method)
   (scan-method method class)
   (let ((methodDic (methodDic class)))
     (attach method methodDic)
     (attach sel methodDic)))

(defun remove-duplicates (lst)
    (do ((l lst (cdr l))(res ()))((atom l) (nreverse res))
        (unless (member (car l) res) (newl res (car l)))))
(defun herit-i-v (supclass i-v)
   (remove-duplicates (append
      (apply 'append (mapcar (lambda (Ci) (i-v (i-v* Ci))) supclass))
      i-v)))

(make-keywords '(name supers i-v keywords methods))
(setq Class '#(Class Class (Object)
   (isit name supers i-v keywords methods)
   (:name :supers :i-v :keywords :methods)
   (new          (lambda (self . i-v*) (send (makeInstance name) 'initialize i-v*))
    initialize   (lambda (self iv*)
                   (initIv self (cdr (i-v (i-v* isit)))
                               (scan-match (keywords (i-v* isit)) iv*))
```

```
                (setq i-v (herit-i-v supers i-v))
                (setq methods (scan-methods methods self))
                (setq keywords (make-keywords (cdr i-v)))
                (rewrite self isit)
                (set name self) )) ))

(send Class 'new  :name 'Object
  :i-v      '(isit)
  :methods
  '(class      (lambda () isit)
    class?     (lambda () (send (i-v* isit) 'metaclass?))
    metaclass? (lambda () (memq 'supers i-v))
    ?          (lambda (iv) (i-v* iv))
    ?<-        (lambda (iv v) (set iv v))
    error      (lambda -msg- (print "selecteur inconnu  " sel " de la classe " isit)
    initialize (lambda (iv*)
                 (initIv self (cdr (i-v (i-v* isit)))
                   (scan-match (keywords (i-v* isit)) iv*))) ))

(send Class 'new :name 'Class
 :supers '(Object)
 :i-v    '(name supers i-v keywords methods)
 :methods
  '( basicnew      (lambda () (makeInstance name))
     new           (lambda i-v* (send (send self 'basicnew) 'initialize i-v*))
     initialize    (lambda (iv*)
                     (run-super)
                     (setq i-v (herit-i-v supers i-v))
                     (setq methods (scan-methods methods self))
                     (setq keywords (make-keywords (cdr i-v)))
                     (set name self))
     i-v           (lambda () i-v)
     subclassof    (lambda () supers)
     methodsDic    (lambda () methods)
     name          (lambda () name)
     selectors     (lambda () (selectors methods))
     methods       (lambda () (methods methods)) ))

; Static access to instance variables

(defun memdic (var l-var l-val)
  (when l-var
    (if (eq (car l-var) var) l-val
        (memdic var (cdr l-var) (cdr l-val)))))
(defun fnref (slot obj) (car (memdic slot (i-v (class-of obj)) obj)))
(defun setref (slot val obj)
  (rplaca (memdic slot (i-v (class-of obj)) obj) val)
  val)
(dmd ref (slot obj) '(car (memdic ,slot (i-v (class-of ,obj)) ,obj)))
(defun scan-method (l classe)
 (scan-parameters l (cadr l))
 (when (cddr l) (scan-body l (i-v classe) classe)))

(defun scan-body (l i-v classe)
  (let ((parameters (cadr l))
        (super-i-v (i-v (class-of classe))))
    (setf (cddr l)
          '((smacrolet
              ,(append
                 (mapcar #'(lambda (slot) (list slot slot))
                         parameters)
                 (mapcar #'(lambda (slot)'(,slot (lambda(var) '(ref ',var self))))
                         i-v)
                 (mapcar #'(lambda(slot)
                             '(,slot (lambda (var)'(ref ',var (class-of self)))))
                         super-i-v))
              ,@(cddr l))))
    l))

(send Class 'new :name 'Class ...)
(send Class 'new :name 'Object ...)

(defun send (obj -selector- . -args-)
   (apply (lookup -selector- (class-of obj) obj) obj -args-))
```