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 ac-
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))
```

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 class 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. (+ (S Metaclass) New 'ListMetaClass '(Class))
2. (+ (S Class) New 'Point 'Object)
3. (- ($ ListMetaClass) New 'Book)
4. (- ($ Point) New 'a_point)
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):

```
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. 36 [1]).
```

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.

```
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 is a subclass of the abstract class Class. Then, all Smalltalk metaclasses are subclasses of Object, 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."
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 a_point, 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 instances1 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.

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:

\[
\text{object} = \langle \text{data}, \text{procedures} \rangle
\]

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:

\[
\text{send object selector Args1 ... Argsn}
\]

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:

\[
\text{class variable } [\text{an object}] = \text{instance variable } [\text{an object's class}]
\]

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.

1To easily distinguish them, we use upper-case initial letters for classes and metaclasses plus bold letters for metaclasses, and lowercase letters for terminal instances.
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 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 analogous 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>.

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 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) \Rightarrow #(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 semantics 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-v1 :i-v2 ... :i-vn) \Rightarrow 
#(Aclass i-v1* ... i-vn*)

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 | ^Point |
| :name | ^Point |
| :supers | ^Object |
| :i-v | ^[x y] |
| :methods | || |
| :x | (λ (x) x) |
| :y | (λ (x y) (setq x y) self) |
| :init | (λ (x y) (setq x 20 y 30) self) |
| :display | (λ () |
| | (format () |
| | (concatenate "*'I 1 1 1 "D") |
| | )>() |
| | )(() |

Then we create instances of Point, using the same new message:

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

2Each 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-v1* ... i-vn*] |
| i-v [class-name] instantiate | #([class-name i-v1* ... i-vn*]) |

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

(defun makeInstance (aclass |
| (icons 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 and a_point2 are equal. Keywords may be also omitted, in this case the associated instance variable is bound to the nil default value (e.g. a_points, a_point4 and a_point5 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 : (\(X\) (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 instantiation 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 :

\[
\text{init name supers i-v methods (new (self . i-values)) }
\]

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

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 manually the skeleton of Class. If we represent objects as lists, we will use the skeleton :

\[
\text{(setq Class '(
Class
Class (Object) (i-v* methods)
(new (self . i-values))
initialize (\(X\) (i-v* init self . . .) (setq i_v (heri-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] :

\[
\text{(send Class 'new :name 'Class :supers '(Object) :i-v '(name supers i-v methods) :methods '())}
\]

name 0 0 name1
super* 0 0 supers)
i-0 (A 0 i-4
metbodsDic (A () methods)
. . . I..
understand (A (selector method) (defmethod self selector method)) selectors (\(X\) (selectors methods) ) )

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

? (send a_point 'i-values)
\#((Point 20 30)
? (send Point 'i-values)
\#((Class Point (Object) (init x y) (x \(X\) display \(X\))
? (send Class 'i-values)
\#((Class Class (Object) (init name supers i-v methods) (new . .))

160 OOPSLA '87 Proceedings October 4-8, 1987
The definitions of the methods show 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 (make-in-stance) 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.

The reflective definition of ObjVlisp allows to use only one allocator - the base new - and nevertheless to exploit 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 instantiation link (the umbilical cord) between a class and its instances.
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. A metaclass 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 cvi variables:

```
(send Class 'new
  :name 'Ametaclass
  :i-v '(cv1 ... cvn)
  :supers '(Class)
  :methods '(...)
)
```

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

```
(send Ametaclass 'new
  :name 'Aclass
  :i-v '(iv1 ... ivn)
  :supers '(...)
  :methods '(...)
  :cv1 cv1
  ...
  :cvn cvn
)
```

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 '(newinit (A 0
                      (send (send self 'new) 'init)))
            char (A () char)
            char: (A (newchar)
                      (fseta char newChar)
                      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 MetaPoint 'new
  :name 'DefaultPoint
  :supers '('Object)
  :i-v '(x y)
  :methods '((init (A () (setq x 40 y 12) self)
                x (A () x)
                y: (A (nx) (setq x nx) self)
                display (A ()
                          (format ()
                           (catenate "=" x "D*" char))
                          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:

```
Class
  (isit name supers i-v methods)
Object
  (isit x y)
MetaPoint
  (isit name supers i-v methods char)
Point#
  (isit x y)
DefaultPoint
  (isit x y)
Point@
  (isit x y)
```

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:

```
Point#
  (isit x y)
DefaultPoint
  (isit x y)
Point@
  (isit x y)
```

```
```

162 OOPSLA '87 Proceedings October 4-8, 1987
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.

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, undefined for a polygon - we use the nSides class variable. To simplify the next figure, the inherited variables are not drawn:

```
Hexagon ... Polygon .... Square

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)
  :methods '
    (sons (lambda (lambda (cdr sons))
      new (lambda (lambda (cons sons (run-super) ()))))
    mapsons (lambda (unaryS)
      (mapc
        (lambda (send rec unaryS))
        (send (send sons))
        unaryS))))
```

The sons method gives access to the sons class variable and the mapsons method distributes an unary message (without argument) 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)
ok
```

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 an 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)* [8]."

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) parametrising 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 distinct 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 Mark, 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


Smalltalk-80 and Loops are trademarks of Xerox Corporation.

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:

\[
\begin{align*}
&\textbf{(a)} (x) \\
&\textbf{(a)} (\text{ref}'x\,\text{self}) \\
&\textbf{(a)} \,() \\
&\textbf{(a)} \,() \\
&\textbf{(format)} \,(\ldots \,\text{char}) \\
&\textbf{(format)} \,(\ldots \,\text{ref}'\text{char}\,\text{class-of}\,\text{self})) \\
&\textbf{(a)} \,(nx) \\
&\textbf{(a)} \,(nx) \\
&\textbf{(setq} \,\text{x:}\,\text{nx}) \,\text{self} \\
&\textbf{(setf} \,(\text{ref}'\text{x\,self}) \,\text{nx}) \,\text{self}
\end{align*}
\]
(setq 1-v (herit-i-v supers 1-v))
(setq methods (scan-methods methods self))
(setq keywords (make-keywords (cdr 1-v)))
(rewrite self isit)
(set name self))
)

(send Class 'new :name 'Object )
:i-v ' (init)
:methods
'class (lambda () (init (1-v isit) 'metaclase?))
?c- (lambda (iv v) (set iv v ))
error (lambda -seg- (print "selecteur inconnu ", sel * de la classe * isit)
initialize (lambda (iv v ))
 (initiv self (cdr (1-v (1-v+ isit))))
 (scan-match (keywords (i-v+ isit)) iv)))
)

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

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

(setq (cddr 1)
'(('macrolet
 (append
 (mapcar '&(lambda (slot) (list slot slot))
 parameters)
 (mapcar '&(lambda (slot)('.slot (lambda (var) '('.ref '.var self)))
 1-v)
 (mapcar '&(lambda (slot)('.slot (lambda (var) '('.ref '.var (class-of self))))
 super-i-v))
 .0(cddr 1))))
)

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

; Static LCC.BI to Imtanc. variables
(defvar nemdic (var l-val 1-val)
 (when l-var
 (if (eq (car l-var) var) l-val
 (nemdic var (cdr l-var) (cadr l-val)))))
(defvar furef (slot obj) (car (nemdic slot (1-v (class-of obj)) obj))
(defvar setref (slot val obj)
 (replace (nemdic slot (1-v (class-of obj)) obj) val)
 (and ref (slot obj) '('.car (nemdic .slot (1-v (class-of obj))) .obj))
 (defun scan-method (1 classe)
 (scan-parameters 1 (cadr 1))
 (when (cadr l) (scan-body 1 (1-v classe classe))))

(defun scan-body (1-classe)
 (let ((parameters (cadr 1))
 (super-i-v (1-v (class-of classe)))))

(defvar nemdic (var l-val 1-val)
 (when l-var
 (if (eq (car l-var) var) l-val
 (nemdic var (cdr l-var) (cadr l-val)))))
(defvar furef (slot obj) (car (nemdic slot (1-v (class-of obj)) obj))
(defvar setref (slot val obj)
 (replace (nemdic slot (1-v (class-of obj)) obj) val)
 (and ref (slot obj) '('.car (nemdic .slot (1-v (class-of obj))) .obj))
 (defun scan-method (1 classe)
 (scan-parameters 1 (cadr 1))
 (when (cadr l) (scan-body 1 (1-v classe classe))))

(defun scan-body (1-classe)
 (let ((parameters (cadr 1))
 (super-i-v (1-v (class-of classe)))))