Object-Oriented Programming in Lisp

Nobuyasu OSATO, Hiroshi G. OKUNO, and Ikuo TAKEUCHI
Musashino Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation

1. Introduction

Many languages such as Simula, Clu, and Smalltalk-80, support object-oriented programming which leads to the modularity of programs. Some Lisp systems also support object-oriented programming, e.g., Flavor in Zenata and Loops in Interlisp-D, but they adopt different approaches. In this paper, TAO's approach to object-oriented programming is described and then approaches in other Lisp's are also discussed.

2. Terminology

An object has a set of internal states which are contained in its instance variables. Instance variables can neither be referred to nor be changed by outer procedures unless message protocols are used. An object also has a set of procedures called methods which represent the object's behavior responding a message. To have an object perform some operation, a message should be sent to the object. An object returns a value responding to a message. The object sent a message is called receiver. An expression to send a message to a receiver is called message passing. A message comprises a message selector and message arguments (optional). Message selector serves as a key for the corresponding method which takes the message arguments as its (procedure) arguments. Note that each method has an implicit argument "self" which denotes the receiver of the message.

The interface between two objects is determined by messages that can be sent between them. This guarantees a highly modular design of programs, which is the important advantage of object-oriented programming paradigm.

Similar objects are categorized into the notion of "class". A class can be deemed as a stencil for individual object. Creation of an object from the class description is called instantiation. Therefore, an object is an instance of a class. A class may have some variables which are called class variables. Class variables are accessible to all objects within the class and are used as common variables among the instances of the class.

Suppose that a class, say A, has the same properties of other class, say B, and suppose that A has some additional properties of its own. In this case, class A can be defined using B as its part. A is called B's subclass, and B is called A's superclass. Note that while a superclass of a class can be thought as an abstraction of the subclass, the superclass can also be deemed as a component of its subclass, which specifies the subclass's property in part. In this sense, a superclass is referred to as a component class. If a class can have at most one superclass, then the sharing of common properties among classes is called a single inheritance or hierarchical inheritance. The capability that a class can have several superclasses is called multiple inheritance.

A class inherits the superclasses' instance variables and their methods. However, if instance variables or methods of the same names are redefined (or overridden) in a class, the superclasses' instance variables or their methods are not inherited to the class.

Suppose that two classes, say A and B, have some common properties and differ only in some part. In this case, the user can define a new class, say C, which has just the shared properties of A and B. After doing this, A and B can be redefined as the subclasses of C. If instantiation of the class C have not any meaning, C is called an abstract class.

If one thinks every object including class as an instance of a class, one should conceive a class as an instance of another class. This class is called metaclass, and metaclass itself should be an instance of another class and so forth.

If message selectors conflict in the inheritance network, the first method in the depth-first left-to-right network traversal is selected in normal situation. However, the user may be able to specify a particular overridden method in some superclass (super-method invocation). Or the user may be able to combine methods of the same message selector (method combination).
3. Object-oriented programming in TAO

3.1 User defined object

Every primitive data types in TAO, behaves as an object. Primitive objects, such as numbers, identifiers, strings and so forth are system defined objects. In addition, the user can define his own object, which has a data type name "udo" (user-defined object). Udo's are marked by their tag and processed efficiently by firmware.

TAO has no notion of metaclass. Classes need not be objects. Methods which correspond to class methods in Smalltalk-80, such as instance creation etc., are defined in Lisp environment.

A class can be defined in the following form:

```
(defclass class-name
  class-variable-list
  instance-variable-list
  lopt superclass-list
  irest defclass-options )
```

where, class-name is an identifier which is the name of the class. The class-name has the class definition in its property list. Class-variable-list is a list of class variable names. Instance-variable-list is a list of instance variable names with optional default values. The optional superclass-list is a list of superclass names, since multiple inheritance is available in TAO.

Inherited instance variables are copied to this class as if they had been listed in the instance-variable-list. On the contrary, the class variables are not copied. However, the superclasses' class variables can be accessed from this class's methods through the class inheritance network. Such class variables are looked up by traversing the network in depth-first left-to-right order. Class variables must be referred to by using a special form as in Loops.

The irest (so-called nospreading lambda) argument may contain various optional specifications of this class. It may include default creation of some kinds of methods, for example, to get the value of a variable or to set a value to a variable, and so forth. It also includes some declaration of this class's property.

3.2 Message passing form in TAO

A message passing form is an s-expression of the form:

```
(receiver message-pattern arg ... )
```

This form is interpreted as a message passing form when its car turns out not to be a function, array, macro, etc. (Otherwise, it is interpreted as a usual function form, of course.) An alternative message passing form is:

```
(receiver message-pattern arg ... )
```

A bracketed list is completely the same as a normal parenthesized list except that its first cell is pointed to with some special tag. Only "eval" switches its action on the tag.

Bracketed form is essential when the receiver itself is a function (applicable object in TAO terminology), which is, of course, a primitive object in TAO. It also serves as a declaration for compiler optimization. A bracketed form \(A . B\) is used to represent \(\text{cons } A B\), which cannot be written as \(\text{cons } A B\) obviously. (Note that the dot in the bracketed form is read as an identifier.)

Message-pattern is either an identifier or a list. If the message-pattern is an identifier, it is called an id-message. A method which corresponds to an id-message is called id-method. Similarly, the terms "list-message" and "list-method" will be used. The role of list-message is the same as that of id-message except that it is used as a pattern for unification in its method search. The details will be omitted here.

3.3 Infix notation using the message passing form

Message passing form in TAO enables the user to express an ordinary arithmetic infix notation like \(A + B\). In addition, nested message passing forms can be flattened into a single list, if the first form is the receiver of the second form and the second form is the third form and so forth. For example, it is possible to replace the ordinary Lisp form:
by a form easier to read:

\[(A + B - C + (D * E) + F)\]

In TAO, infix notation together with named cell facility which enables a normal mathematical representation of function like "f(x)" enhances the program readability.

3.4 Method combination in TAO

In TAO, methods of the same message selector are combined statically like flavor of Zetalisp. Hence, the fuss of combining methods in specified way is done only once. A combined method is itself an independent method and is registered in the id- or list-message vector.

4. Implementation of object-oriented programming in TAO

4.1 Class description

A class is represented in a vector of size 5, called class-vector, whose title is "class". It has ten slots for internal information as shown in Figure 1.

1. id-message vector

Each element of this vector is a pair of id-selector and corresponding method which is defined in the class or inherited from its superclass. The pairs are arranged in order of selector's addresses so that methods can be looked up by binary search strategy. In this vector, methods get value of an instance variable or set the value are represented by integers (positive for get, negative for set) which denote the relative position of the instance variable in the instance variable vector described below.

Id-message vector is constructed when the first id-message is sent to an instance of this class. This is because the system allows incremental definition of methods. When all of the methods are defined, all id-message selectors are ready be sorted. Complete information for creating the id-message vector is maintained in the defclass-skeleton slot of the class-vector, which will be discussed later.

Id-message vector is modified when some change happens in the class or its superclass. This modification, however, only shows that some change has happened. Method body is actually redefined (or recompiled) when the first message for this method is sent.

2. list-message vector

Each entry of this vector is a pair of list-selector and its corresponding method. Unlike id-message vector, the order of the entries are determined by the definition order. Methods are looked up from top to bottom sequentially in this table. The creation and modification of this vector is the same as id-message vector.

3. super-id-message vector

This table holds the methods of the superclasses which are not inherited to the class because of overriding definitions of the message selectors of the same name, but are actually used in super-method invocation. To access directly to a particular superclass's method, a compound message is used, which looks like, for example, "foo.initialize", where "foo" is the name of the particular superclass (called class-name prefix) and "initialize" is the selector in the superclass (selector suffix).

Note that the "foo.initialize" is itself a single identifier and becomes the selector of this (not "foo"'s) class. Like id-message vector, the entries of super-id-message vector is also sorted by the identifiers' addresses for binary search.

The form to send a compound message to self is:

\[(super* class.message arg ... )\]
where super* is a Lisp function which sends class: message to self. Methods are sought in the super-id-message vector. If class: message is not found in this vector, then the compound identifier is decomposed and the corresponding class's defclass skeleton is sought. If the desired method is found, it is redefined in this class and added to the class's super-id-message vector.

Another form:

(super message arg ...)

is just the counterpart of Smalltalk-80's super-method invocation. Note that, in this case, the message doesn't have superclass prefix.

This vector has initially no entries and is augmented whenever a new (yet unknown) compound message is sent to an instance of this class. In this sense, this vector is a kind of method cache.

(4) super-list-message vector

This is the list-message counterpart of the super-id-message vector. A list-message which specifies a class with which search begins can be sent to self by evaluating a Lisp form:

(super-unify* class list-message arg ...)

This message is registered in the vector by a list key whose car is "class" and cdr is "list-message". Another Lisp form:

(super-unify list-message arg ...)

works almost the same as super-unify* except that methods are sought starting from the direct superclass of the class.

(5) id back pointer

Every class vector knows for what identifier it is defined via id back pointer.

(6) version number

This number shows the version of the class vector associated with a particular identifier. Class vector is newly created, and thereby the version number is updated, every time when names or number of the instance variables are changed by redefining the class or its superclass. Note that the old class vector is disassociated with the corresponding identifier, though the id back pointer of the vector is still unchanged.

(7) class variable vector

Each element of this vector is a pair of a class variable and its value. The vector consists of only the variables which are declared in this class's definition. Other class variables inherited from superclasses are located in each superclass.

(8) property list

A class can have a property list. Properties can be got or put by the ordinary get and putprop functions. Various properties such as abstractness, information for documentations and so forth are stored in this list.

(9) defclass skeleton

This slot keeps the complete description of the class definition as follows:

(i) a list of direct superclass names explicitly specified here
(ii) a list of class variables
(iii) a list of instance variables excluding the inherited ones
(iv) options which is the first argument of the defclass form
(v) a list of id-messages explicitly defined here
(vi) a list of list-messages explicitly defined here
and so forth.

(10) make-instance skeleton

This is a list of all instance variables optionally paired with default values, which is used at an instance creation of the class. Make-instance skeleton is created at the time of first instantiation of this class.

<table>
<thead>
<tr>
<th>vtitle: class</th>
<th>vsize: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>id-message-vector</td>
<td>version number</td>
</tr>
<tr>
<td>list-message-vector</td>
<td>id backpointer</td>
</tr>
<tr>
<td>super-id-msg-vector</td>
<td>super-list-msg-vect</td>
</tr>
<tr>
<td>class-var-vector</td>
<td>property list</td>
</tr>
<tr>
<td>defclass skeleton</td>
<td>make-inst skeleton</td>
</tr>
</tbody>
</table>

Fig. 1 Class vector

4.2 User defined object (udo)

A user defined object is a kind of vector whose tag is "udo". It contains all instance variable values and information necessary to respond to messages. A udo consists of the following elements. (Figure 2 depicts the structure of udo.)

(1) class vector (in the position of vtitle)
(2) instance variables

Each element of this vector is a pair of instance variable and their value. The methods of a class can be deemed as function closures with a common set of variables. However, the instance variables cannot be accessed from the methods outside the class and its subclasses.

<table>
<thead>
<tr>
<th>class vector</th>
<th>vsize: n</th>
</tr>
</thead>
<tbody>
<tr>
<td>value 1</td>
<td>variable name 1</td>
</tr>
<tr>
<td>value 2</td>
<td>variable name 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>value n</td>
<td>variable name n</td>
</tr>
</tbody>
</table>

Fig. 2 Udo structure

4.3 Inheritance network bookkeeping

Every class name identifier has a property "component-of-what". The property value is a list of all direct subclasses of this class. The value is put at the first creation of the class's instance. If this class is redefined, component-of-what of its old direct superclasses should be updated first and then the effect of the redefinition is rippled down to subclasses recursively.

4.4 Method lookup strategy

TAO uses an all-in-one method lookup table according to the following consideration. As described in the preceding sections, id-method table is sorted by the selectors' addresses. Suppose an id-message vector has N elements. In binary search, only one element can be found in the first trial. Two elements can be found in the second. Similarly, $2^\left\lfloor \log_2 N \right\rfloor$ entries can be found in the floor($\log_2 N$)-th trial, and $N - 2^\left\lfloor \log_2 N \right\rfloor$ entries can be found in the ceiling($\log_2 N$)-th trial. Thus, average seeking cycles is calculated by
\[ S = \sum_{k=1}^{m} \left( k \cdot 2^{(k-1)} + m' (N - 2^m) \right) \]

where, \( m = \text{floor}(\log N) \) and \( m' = \text{ceiling}(\log N) \).

Single (non-compound) id-method is either from the receiver's class or from one of its superclasses. If inherited methods are invoked less frequently than its own methods, two level method table may be desirable, since the smaller becomes the table size, the faster can be found a method.

The statistical data on a Smalltalk-80 implementation at Hewlett-Packard shows that more than half methods (56%) of messages sent are found in that class. This suggests the effectiveness of the two level method table strategy.

Suppose that the table is divided into two subtables; one for its own methods and the other for inherited ones. The average seeking time for the former table is given by the above formula. Normal method lookup starts from the first table. If the method is not found in it, the second table is inspected. Thus, the average seeking time for the second table is calculated by adding terms which represent the maximum seeking time for the first table and the traversing overhead between the two tables. Overall average can be calculated by adding two averages taking account of weight factors. Considering the Hewlett-Packard's data and this estimation, two level search table will not be effective unless the frequency of own method invocations is much greater than that of inherited ones. However, this situation doesn't seem to happen in normal programming.

In the current microcoding, one seek cycle takes six microsteps. Since one microstep takes 180 nano-seconds, adding miscellaneous steps, average lookup time is about 4.8 microseconds for 31 id-methods, and about 5.8 microseconds for 63 id-methods, which are considered to be nearly typical.

4.5 System defined object

System defined objects or primitive objects are basic data types in TAO. Primitive operations on these data types, represented by system identifier such as +, -, ..(dot), ...(double dots) etc., are processed in quite an efficient manner, which depends strongly on the microprogrammed implementation. When such a message selector is recognized by "eval", its address is tested if it drops in a specific range. If this is the case, the address is added by an object-dependent bias value to get the corresponding entry address of micro routine and the control is transferred to it.

Primitive data types can be also deemed as classes in TAO. Hence, of course, the user can add new methods to them, and can override or mix methods except for the system defined messages. For example, on the primitive class integer, the user can define a new method, say, a method which respond to a query message "Is-it-a-prime-number?" by returning t or nil. Since all primitive objects do not take udo structure, class descriptions of these objects are maintained in fixed system data area.

5. Comparative study on object-oriented programming in Lisp

5.1 Object oriented programming in MacLisp

MacLisp provides only a flavor of object-oriented programming. That is, it is possible to localize functions to some classes, but provides no built-in mechanism of inheritance.

5.2 Flavor in Zetalisp

Object-oriented programming in Zetalisp is called the Flavor system, or simply Flavor. NIL (New Implementation of Lisp, currently available on VAX/VMS) also provides a subset of Flavor. A flavor is a class analog of Flavor. In Flavor, an abstract class is called mixin flavor. No class variable is attached to a flavor.
Flavor is the first system which implements full-fledged method combination. This method combination is considered as the central issue in the design of Flavor. If the component methods don't interact each other, the behavior is easily performed by combining those component behaviors, in other words by invoking the methods corresponding to them one by one in an arbitrary order. The combination of interacting behaviors, however, requires a modularly interacting mechanism on the basis of orthogonal treatment of each module (flavor).

The type of method combination is classified into daemon combination and non-daemon combination. For daemon combination, there are three types of method: primary, before, and after. For non-daemon combination, primary methods are executed in OR, AND, MAPCAR, or REVERSE-MAPCAR manner. Wrapper provides the capability of combining methods in an arbitrary manner. Wrapper is implemented by macro in Flavor.

In Flavor, message passing form is a function call:

```
(funcall receiver message arg ...) or
(send receiver message arg ...)
```

that is, flavors are regarded as function object (extension of function closure). This is an outstanding difference from other object-oriented programming languages. Note that message is evaluated. The another difference is that Flavor does not support super-method invocation.

Flavor's inheritance network is flattened to make a template for instance creation and variable access of method, which TAO inherits as has been shown. The association table of message selector and method is maintained and accessed by hashing mechanism. This flattening plays an important role in executing combined methods. In the current implementation, this is performed at the first instantiation time.

5.3 Object-oriented programming in Loops (Interlisp-D)

Loops adds object-, data-, rule-oriented programming paradigms to the procedure-oriented programming paradigm in Interlisp-D. The central issue of Loops design is that a programming language should provide a set of programming paradigms of which one can select a subset that best matches structure of the problem, especially on knowledge based systems. Object-oriented programming in Loops is similar to that of Smalltalk-80 with multiple inheritance as an additional feature. Message passing form in Loops is as follows:

```
(send receiver message arg ...) or
(<- receiver message arg ...)
```

where message is implicitly quoted. As shown below, instance creation and class definition are also invoked by message passing. Therefore, Loops has a metaclass concept. In Loops, an instance can be named, which is shown in the last form.

```
(<- $Class New className superList)
(<- (<- $Ship New) SetName instanceName) or
(<-New $Ship SetName instanceName)
```

where $ is a read macro to obtain an internal pointer to the object, and New and SetName are message selectors.

Both class variables and instance variables may have property list. Instance variables are accessed via @ read macro, and class variables via @@ read macro. How to access instance variables is summarized below. As for class variables, read @@ for @, GetClassValue for GetValue, and PutClassValue for PutValue. Note that it is impossible to access instance variables by their names only. In that sense, they are not genuine variables.

<table>
<thead>
<tr>
<th>Form</th>
<th>Macro expanded form</th>
</tr>
</thead>
<tbody>
<tr>
<td>@X</td>
<td>(GetValue self 'X)</td>
</tr>
<tr>
<td>@X(Obj X)</td>
<td>(GetValue Obj 'X)</td>
</tr>
<tr>
<td>@X(Obj X P)</td>
<td>(GetValue Obj 'X 'P)</td>
</tr>
<tr>
<td>@X &lt;- newValue</td>
<td>(PutValue self 'X newValue)</td>
</tr>
<tr>
<td>@X(Obj X) &lt;- newValue</td>
<td>(PutValue Obj 'X newValue)</td>
</tr>
<tr>
<td>@X(Obj X P) &lt;- newValue</td>
<td>(PutValue Obj 'X newValue 'P)</td>
</tr>
</tbody>
</table>

({}
Data-oriented programming means that if a variable is accessed (modified), Get functions (Put functions) associated to the variable are triggered. A variable with such triggered functions is called active value. An active value is specified as an initial value of instance variable in the definition of class. The format of an active value is:

```lisp
#{localState getFn putFn}
```

where `localState` may be an identifier or another active value. If an active value is nested, associated Put functions are triggered in sequence from the outermost to the innermost and associated Get functions in opposite order. This mechanism resembles method combination. This mechanism is implemented by using property list of variables.

Other method combination forms Loops provides are:

```lisp
(DoMethod receiver message class arg ...)
(TryMethod receiver message class arg ...)
(ApplyMethod receiver message argList class)
```

where `class` specifies the class from which the method is sought. If `class` is nil, the class of receiver is assumed. Although these method combinations are limited as compared with Flavor, Flavor-like method combination can be implemented by using Interlisp functions.

Super-method invocation in Loops is:

```lisp
(<-Super receiver message arg ...)
```

If receiver is self, it is equivalent to TAO's message-passing form:

```lisp
(super message arg ...).
```

Note that the receiver may be any object other than self unlike Smalltalk-80 and TAO.

Values and property lists of instance variables are not set until they are actually accessed. This is a simple consequence of the fact that they are implemented by property list of native Interlisp.

6. Concluding remarks

TAO attempts to union the features of Smalltalk-80 and Flavor, since each of them has its own useful features which the other has not. The data-oriented programming paradigm in Loops can be easily and effectively implemented in TAO. Currently, primitive message passing to primitive objects such as \(+\) to integers is now running in microcode and proves to be as fast as equivalent Lisp forms such as \((+ x y)\). Microcoding for general message passing to user defined objects is now well under way.

Acknowledgments

The authors gratefully acknowledge Dr. K. Tsukamoto for his continuous encouragement, members of NUE group the authors are collaborating with for their valuable discussion, and Dr. N. Suzuki for his helpful discussion and valuable information.

References

[GOL83] A. Goldberg and D. Robson. Smalltalk-80: The Language and its implementation. Reading, Massachusetts, Addison-Wesley, 1983.