Object-Oriented Programming in Multiple-Paradigm Language TAO and Its Implementation

NOBUYASU OSATO* and IKUO TAKEUCHI**

The language specification of object-oriented programming in multiple programming paradigm language TAO and its performance evaluation are described. TAO's goal is to provide a high performance programming environment for artificial intelligence research. TAO's central part is microprogrammed on a dedicated symbol manipulation machine called ELIS. One of the main design principles of TAO is to allow the programmer to choose a suitable programming paradigm for his objective. To achieve this principle, TAO incorporates multiple programming paradigms such as logic programming, object-oriented programming and so forth into its basic Lisp programming. This programming paradigm fusion is accomplished within the kernel of the language. This paper discusses the TAO's object-oriented programming: its implementation technique in detail and performance evaluation. It demonstrates that TAO's object-oriented programming is efficient enough and by no means inferior to its Lisp programming paradigm from the viewpoint of practicality. The validity of the implementation method is also discussed by analyzing a couple of practical application programs.

1. Introduction

TAO/ELIS is a symbol manipulation system developed at NTT Electrical Communication Laboratories. The main motivation behind its development was to provide a powerful and flexible programming environment for artificial intelligence. The hardware ELIS [1] and the software TAO [2] were simultaneously and interdependently developed. TAO was designed as a Lisp-based kernel language to establish a high performance programming environment. It aims at accomplishing efficient interpreted execution, supported by the performance of the ELIS hardware [3]. TAO's most speed-critical parts, e.g. eval, the primitive functions and the basic mechanisms, are completely microcoded on ELIS, which has a huge amount of writable control store (WCS). An efficient interpreter is quite preferable in the development cycle, since tests and debugs require a lot of information during program execution.

One of TAO's design principles is to allow the user to choose his/her programming paradigm so that the choice fits the problem to be solved. The syntax and semantics of Lisp are quite simple, and they enable various programming paradigms to be incorporated into the language. TAO achieves its paradigm fusion under Lisp's simplicity and extensibility. TAO's multiple paradigms include logic programming, object-oriented programming [4, 5] and others as well as conventional Lisp programming, which is based on Common Lisp. In particular, object-oriented programming is one of the main paradigms of TAO [6]. This paper discusses the TAO's object-oriented programming language specification, implementation techniques and its performance evaluation.

2. Object-oriented Programming in TAO

2.1 Objects and Messages

In object-oriented modeling, logical entities in the subject are deemed to behave autonomously. These entities are called *objects*. There are two kinds of objects in TAO. One is the primitive objects such as integers, strings, ids (which stands for identifiers) and so forth. The other is the user defined object, abbreviated as udo. The udo has some data slots called *instance variables*, each of which consists of a pair of a slot name and its value. Objects with the same structure and the same behavior are grouped under the concept of *class*. Each object belongs to a certain class and is called an *instance* of the class.

In object-oriented programming, the computation proceeds by passing *messages* among objects. A message passing occurs when a specially devised *message passing form* is evaluated. Each object has its own internal procedures called *methods*. When an object receives a message, it chooses the corresponding method and exe-

This is a translation of the IPSJ Best Paper Award paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 30, No. 5 (1989), pp. 596-604.

*NTT Human Interface Laboratories, 3-9-11, Midori-cho, Musashino-shi, Tokyo 180, Japan.

**NTT Basic Research Laboratories, 3-9-11, Midori-cho, Musashino-shi, Tokyo 180, Japan.

cutes it. A message passing form specifies its *receiver* object, the message name (id) to select a method. The id is also called a *message selector* or, simply, a *selector*.

The general message passing form in TAO is:

[receiver-object message argument . . .]

where, [ ] is a kind of list tagged as a special data type *bracket*. This form is evaluated in the following manner:

(1) The car of the form, *receiver-object*, is evaluated.

(2) If the resulting value is an object, the cadr part *message* of the form is deemed as the message selector and the corresponding method is sought. The *message* is not evaluated.

(3) If a method is found, it is invoked with the trailing argument *argument*.

(4) The form evaluates to a value as an ordinary Lisp function does.

A message passing form can appear at any position of a program. The resulting value is used in the Lisp context. The syntax of a message passing form differs from that of an ordinary Lisp form, which has a prefix operator, while a message passing form has an infix operator. The infix operator enables commonly used representations for arithmetic operations over integers, floats and others such as $[x + y]$ or $[x * y]$.

2.2 Classes and Method Definitions

A class holds its objects' attribute descriptions, methods, *class variables*, which are class's own data slots, and so on. A class itself is not an object. An udo's class is defined by a defclass macro, which has the form:

(defclass *class-name*
class-variable-list
instance-variable-list
{*superclass-list*}
defclass-option . . .)

where, *class-name* is an id: the name of the class. *Class-variable-list* and *instance-variable-list* declare class variables and instance variables, respectively. *Superclass-list* specifies this class's *superclasses* which will be described later. Here, { } denotes that the enclosed variables are optional. *Defclass-option* indicates various optional class specifications. For instance, option :gettable defines instance variable accessing methods automatically.

A method is defined by a defmethod macro, which registers the method to a class. To define a method whose selector is *selector* to a class *class-name*, the macro form:

(defmethod (*class-name* {*method-type*} *selector*)
argument-list
form . . .)

is used. In this form, *method-type* is an optional information used by the *method combination* mechanism which will be discussed later. *Argument-list* is the argument list of this method and plays the same role as the Lisp's function argument list. The trailing argument *form* is the body of this method's procedure.

2.3 Instance Creation

An instance of a user defined class is created by a macro form:

(make-instance *class-name init-option . . .*)

where, *init-option* gives the initial values of the newly created instance's instance variables. Make-instance macro can create only udos. Primitive objects like integers, vectors and so on are created by dedicated functions such as read, +, vcons etc.

2.4 Class Hierarchy and Inheritance

A new class can be defined by modification or combination of already existing classes. The old classes are called *superclasses* of the newly created class, while the new one is called a *subclass*. A class can have more than one superclasses. The new class *inherits* various attributes such as instance variables, methods and others from its superclasses. A partially ordered structure emerges under the inheritance relationship. TAO's inheritance is similar to that of Flavors.

Some superclasses may cause a contention due to their information to be inherited. This necessitates a certain total ordering of the superclasses. TAO determines the superclasses' linearized ordering by traversing them upward depthfirst, starting from the subclass.

2.5 Method Combination

If there are some methods whose selectors are the same among the inherited methods, they are combined to form a new single method using a special technique called method combination. Here, every method in each class is used as a component of the newly created method, which is actually invoked when a message is passed. The methods to be combined are categorized into several types according to their method type as specified in the defmethod forms. A method type defaults to :primary type. There are several types of method combination. When the type of method combination is specified, corresponding method types are determined.

The default method combination type is the one called :daemon. Here, the :daemon combination combines three types of methods: :primary, :after and :before. These methods are incorporated in the newly created method and invoked in the following order.

(1) Each :before method is called in the superclasses' linear order from near to far.

(2) The first :primary type method in the order is called.

(3) Each :after method is called in the reverse order of that of :before methods.

(4) The `:primary` method's value is returned as the combined method's value.

2.6 Super Message Sending

A specific method in a superclass can be invoked in a method. For example, in a method `m` of a class `A`, a method `n` in the `A`'s superclass `S` can solely be called by using a function `super`:

(super *S.n argument . . .*)

where, `S.n` in an id established by concatenating the class name `S` and the selector `n` separated by "." (period). The trailing argument *argument* is passed to the method. The receiver temporarily behaves as if it were the instance of `S`. The class traversal for the method combination starts from the superclass `S`.

3. Implementation

3.1 Defclass Macro and Class Representation

A class is represented by a vector called *class-vector*. The vector is a one-dimensional array of Lisp data. A class-vector is created by a `defclass` macro and is registered as one of the properties of the class-name id. If there already exists a class-vector in the property list of the class-name id, namely, the class is redefined, the old class-vector is replaced by a new one. Since the modification of a class influences its subclasses because of inheritance, the subclasses of the old class are traversed and modified. Although the old class-vector can no longer be accessed from the class-name id, its instances continue to maintain the pointers to their original class-vector. Only the newly created instances point to the new class-vector.

The contents of the class-vector include:

(1) a method table, sorted so that it can be sought with a binary search algorithm,
(2) class-name,
(3) a method table that the function `super` uses,
(4) a property list which is specific to this class-vector,
(5) a class variable table,
(6) information for instance creation,
(7) the original information given in a `defclass` macro,
(8) a hash table for instance variable search speedup.

Class-vectors for primitive data types such as integers, ids and so on are pointed to not only from the property lists of their corresponding class-names but also from a table located in a fixed area of the ELIS's hardware stack for efficiency.

3.2 Udo Representation

Udo is one of the TAO's data types. It is a special kind of vector, which consists of an arbitrary number of pairs of an instance variable and its value slot. Figure 1 depicts the structure of the udo. The udo has a back pointer to its class-vector.

class-vector	2*n* (udo size)
value 1	instance variable 1
. . .	. . .
. . .	. . .
value *n*	instance variable *n*

Fig. 1 Structure of user defined object (udo).

At the moment of the first invocation of `make-instance` in a class, the superclasses of the class are traversed to collect the information to be inherited. The collected data, which is registered to the class-vector, includes instance variable names and their initial values. The udo is created using this data, then the initial values of the instance variables are set.

3.3 Methods and Method Tables

3.3.1 A Method and Its Definition

A Method is represented, in general, by data called applobj, which stands for applicable object. An applobj, which is a TAO's general function object descriptor, is a special vector whose elements describe the body of the procedure, how to bind the arguments, how to construct a stack frame, and so forth. The function body is an S-expression, if it is interpreted, or a block of byte codes, if it is compiled.

An applobj is created by a `defmethod` macro form and is stored in the class-vector for future use in the method combination. After combining method applobjs, the resulting procedure is also represented as an applobj and is registered to the method table together with its selector. A method table is sorted with respect to the internal addresses of the selector ids.

Methods can be redefined or added at any time. Since a method is just data in the class-vector, the class' method repertory can be changed simply by modifying the data. Both old udos and new ones can refer to the same up-to-date method table at a message passing occurence.

3.3.2 Method Table Construction

The method table is created at the first moment when an instance of its class receives a message. When a message is sent to an object of the class for the first time, an internal function is invoked to construct the method table of the class. At this moment, the table has only empty slots for each method applobj. After the construction is completed, the method table is sought using a binary search algorithm. In addition, each applobj entry of the method table is created on demand at its first invocation time. This internal function's behavior is as follows.

(1) Starting from the class, it traverses the superclasses to collect method selectors to make a list

of method table entries,

(2) constructs a vector for the method table to accommodate the selectors collected in step (1) and the corresponding method applobjs. Each method applobj slot is set empty, indicating that the applobj should be created later,

(3) registers this vector to the class-vector,

(4) creates only the method applobj just about to be invoked. To be precise, it traverses the superclasses to collect all corresponding applobjs, combines them, and fills the slot with the newly created applobj, and

(5) invokes the applobj.

For each message, when it is sent for the first time, i.e., when the corresponding method table slot is empty, steps (4) and (5) are processed. Since checking for the method table existence and empty applobj is performed by multiple-way branching of the ELIS's microcode, there is no additional overhead from this procedure.

This rather complicated process for method establishment may puzzle the reader. This *on-demand* method construction process was adopted because the time required for method construction turned out to be seriously long for large practical applications. In the earlier implementation, the complete method table and its contents were constructed when the first message was passed. When the application size grew such that there were thousands of methods, the waiting time for the method construction increased to more than several minutes, which caused not only a great deal of debugging inefficiency but also the construction of numerous applobjs, many of which were never invoked. The on-demand construction greatly reduces both time and space consumption.

In the program development cycle, methods are redefined or added very frequently on the superclass sides. The modification affects and changes the behavior of the subclasses. Although this leads to a big internal mess, it is logically invisible from the user. As stated above, almost all method constructions are postponed until they are actually invoked. Thus, small and dispersed overheads are required.

3.3.3 Method Combination Implementation

As described before, the method combination constructs a new applobj, which is registered to the method table. Inherited methods are copied and incorporated to this applobj. This eliminates the run-time search overhead for the inherited methods.

As an example of a method combination, the usage of inherited applobjs in the `:daemon` combination is described below. Assuming that the classes have the hierarchy of Figure 2, the methods are defined as:

```
(defmethod (a m) ( ) 'a-m-primary)
(defmethod (b :after m) ( ) 'b-m-after)
(defmethod (c :before m) ( ) 'c-m-before)
```

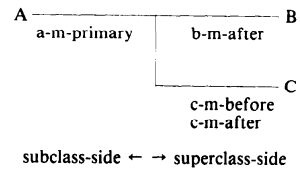

Fig. 2 An example of a class hierarchy and methods. (Methods are represented by their return values.)

```
(defmethod (c :after m) ( ) 'c-m-after)
```

A new method applobj for the method `m` of the class `A` is constructed with the `:daemon` combination and registered to the `A`'s method table. The resulting applobj has the body of:

```
(progi (funcall c-m-before's_method_applobj)
       ^(funcall a-m-primary's_method_applobj)
        (funcall c-m-after's_method_applobj)
        (funcall b-m-before's_method_applobj))
```

where `progi` is a generalized `progn` that returns the value of the form flagged by `^`. `Progi` gives a skeleton to combine the inherited methods. Of course, if there are no `:after` or `:before` type applobjs, unnecessary `progi` or `funcalls` are omitted and the applobj of the inherited method is used.

3.3.4 Sharing Method Applobjs

Since the inherited methods are copied in principle, a great many methods are duplicated. These methods are essentially different because they are called under different environments. In many cases, however, they can actually have the same bodies. To reduce this duplication, TAO uses a hash table to share the same applobjs. The method body S-expression is used as a key to the applobj hashing. When a new applobj is registered, TAO compares the body equality to the already existing ones. The S-expression function body is shared as well as the compiled one, if possible.

This applobj sharing is quite effective in reducing the memory consumption, especially for the large practical applications. According to a measurement of the number of reduced applobjs for some applications, 40% in interpreted programs and from 10 to 40% in compiled codes are reduced. For compiled bodies, identical original S-expression bodies may differ as binary sequences of compiled codes, because their instance variable offsets may vary from one class to another. Thus, the compiled body sharing rate is worse than that of the interpreted one.

3.4 Variables

Since the method is also a Lisp program, many types of Lisp variables, such as local, special and so on, that can be used in an ordinary Lisp program can, of course, appear in it. In addition, instance variables and class variables are available in object-oriented programming.

Instance variables are not bound lexically in a method, i.e., they are free. Thus, the efficiency of their reference plays an important role in the system's performance. in the TAO interpreter, instance variables are hashed into an offset table that indicates the relative positions of instance variables in the udo. This hash table is created at the moment of the first instantiation of the class and is maintained in the class-vector. In the compiled methods, the instance variables' offsets are embedded in byte codes.

Class variables are similar to instance variables in that they can appear only inside a method. Unlike instance variables, class variables are not genuine variables in TAO. They are accessed with a special function `cvar`. Class variables are not inherited but can be accessed through the subclasses' methods. If the class variable is not in the class, the superclasses are traversed to find it.

3.5 Message Passing

User defined methods are processed as follows:

(1) The existence of the method table in the receiver udo's class-vector is checked.

(2) If there is no method table, the internal function described in 3.3.2 is invoked and the table is constructed.

(3) The method table is sought using the `cadr` of the given form as a binary search key. The method table can contain four kinds of entries that switch the system's behavior as follows:

(i) Instance variable accessing method
defined automatically by `:gettable` or `:settable` defclass options. They are used only for instance variable accessing purposes. These methods are represented not by applobjs but by special types of numbers that indicate the instance variables' offsets in the udo. No stack frame is created during execution.

(ii) Method applobj
constructed by a method combination. When this type of method is invoked, a stack frame that holds its environment is created and the body is executed.

(iii) "Empty" flag
indicates that the method is only defined by `defmethods` and has not been invoked yet. If this flag is found, an appropriate applobj is constructed and stored in the table, then case (ii) is performed.

(iv) Otherwise,
the entry is deemed as a *class constant*. In this case, the contents in the slot are returned immediately. A class constant is a special method that returns a constant at any time.

It is important to note that these various data checks are performed in parallel by using the multiple branching mechanism of ELIS microprogramming. Therefore, no overhead is imposed on typical method invocations.

3.6 Messages to Primitive Data Objects

Using message passing forms, arithmetic operations in TAO can be expressed in the infix notation. For instance, in the form $[x + y]$, if x evaluates to a number, this form is interpreted as an addition in the infix notation. If x evaluates to a non-numerical value, e.g., an udo, its method table is searched for the selector $+$. If it is found, the corresponding method is invoked.

The infix arithmetic forms are processed by dedicated micro routines, which bypass method search. In this case, the system calculates the micro routine entry address in the WCS from the selector $+$'s address in the memory by a simple offset computation. The micro body of $[x + y]$ itself is the same as that of the Lisp form $(+ x y)$.

For primary data such as numbers, forms like $[x + y + z]$ are also supported by microcoded interpretation. The first $[x + y]$ is deemed as a message passing form, whose value receives the message "$+ z$". Therefore, this form is equivalent to $[[x + y] + z]$.

3.7 Compiler

Even though TAO emphasizes the performance of interpreted execution, the compiler is useful for reducing time and space consumption. TAO's compiler [7] produces byte codes of a virtual stack oriented machine. Each byte code is interpreted by a microprogrammed interpreter. Since a method is a kind of Lisp function, its compilation is almost the same as that of a Lisp function. The difference is that, since an instance variable reference is free in a method, its access information cannot be acquired until the class structure is determined. So, the instance variables are compiled into a dummy byte code sequence unless they can be determined at the compile time. The instance variable reference is replaced with the correct one when the first message passing occurs and antecedently the class structure has been fixed.

A message passing form is compiled into a code that indicates *a message should be sent*, unless its receiver's data type is declared and its method body is known at the compile time. For instance, the form `[A B C]`, if `A`'s data type is unknown, is compiled into:

```
compiled-code of A
push-constant B
compiled-code of C
send-binary-message
```

where, send-binary-message is a byte code which invokes method search, creates stack frame, and executes the method body. There are two other kinds of byte codes for the message passing form: send-unary-message for unary message passing forms, and send-general-message for general cases.

4. Performance

4.1 Speed of Method Search

TAO adopts a binary search algorithm for method search. There are several implementation alternatives for method tables such as hashing and binary search. The choice is an important factor in the performance. TAO aims at a practical system that can accommodate big applications involving a great many methods. Hashing can be a choice, of course, but it requires an undesirous huge amount of memory for the large number of methods. On the other hand, ELIS microcode enables the efficient implementation of a binary search algorithm.

The binary search microcode has a six-microstep loop for every test in the method table. Since the ELIS's microstep needs 180 *n*sec, this loop consumes about 1 μsec per cycle. Therefore, a search in a method table with N entries takes $\log N$ μsec on average. According to a measurement, the method search requires very low overhead for object-oriented programming in TAO, as will be described later.

4.2 Instance Variable Hash Table

The effect of the instance variable hashing on the interpreter strongly depends on the udo sizes. Among the various actual applications, there exist udos with hundreds of instance variables. This proves the effectiveness of the instance variable hash. Although the hash table is not so effective for small udos with 2 or 3 instance variables, a linear search algorithm is faster than hashing by at most 20%, which makes very little difference.

The accessing speed of Lisp's special variables depends on the existence of the instance variable hash table because of the following reason. In TAO, instance variables are sought right after the local variables. So, if there is no instance variable hash table, special variables are sought in an udo in vain. (Note: TAO adopts deep binding.) Therefore, if the instance variable hash table exists, the system knows quickly whether the free variable in a method is an instance variable or not, and the unnecessary udo search can be avoided.

4.3 Speed Comparison with Lisp by Simple Programs

The evaluation mechanism for a message passing form is essentially the same as that of a Lisp form. The exceptions are the recognition step for the message passing form and the method search. According to our experience in TAO, the processing time for these portions does not play a major role in the performance. This makes TAO's object-oriented programming practical. It is slower than Lisp by only 10 to 20%.

In this section, using recursively defined Fibonacci progression calculation programs as benchmarks, Lisp and object-oriented versions are compared. The function `fibonacci` is defined in Lisp as:

Table 2 Execution time of compiled fibonacci. (unit: *msec*)

n	*Lisp version*	*Object-oriented version (Method table size)*	
		30	100
19	143.2	384.2	413.9
20	231.7	621.7	669.8
22	606.5	1627.7	1753.3
25	2569.1	6894.9	7427.4

Table 1 Execution time of interpreted fibonacci. (unit: *msec*)

n	*Lisp version*	*Object-oriented version (Method table size)*	
		30	100
19	521.8	583.5	612.8
20	844.4	944.3	991.6
22	2210.7	2472.3	2596.0
25	9363.5	10472.1	10997.5

```
(defun fibonacci (n)
    (if (< n 2) 1
        (+(fibonacci (1 − n))
          (fibonacci (− n 2)))))
```

In object oriented programming, the `fibonacci` method is defined in the integer class. A `fibonacci` message is sent to an instance of the integer class, i.e., to an integer. This method is defined as:

```
(defmethod (integer fibonacci) ( )
    (if [self < 2] 1
        [[[self − 1] fibonacci]
            +[[self − 2] fibonacci] ]))
```

where, `self` is an automatically defined local variable bound to the receiver object of this message. The results are shown in Tables 1 and 2.

In these measurements, the method table sizes are assumed to be 30 and 100, and the fibonacci method is found to be in the worst case. In the compiled versions, the variable `n` and `self` are declared fixnums. In the interpreted versions, the performance difference between Lisp and object oriented programming is less than 20%. In a compiler, Lisp versions run 2 to 3 times faster than the object oriented versions do. This is because the compiler for Lisp is optimized while that for the object-oriented version is not. As of the time this paper was being written, the object oriented programming optimizing compiler was not available. The Lisp compiler's optimization includes tail recursion optimization.

4.4 Evaluation through Some Applications

It is very useful to investigate how actual application programs are written in an object-oriented manner. Here, several applications are analyzed and discussed. Three programs: Emacs-like editor ZEN [8], Japanese input system JPRO which is incorporated into ZEN,

Table 3 An evaluation with some application programs.

	TEN	NET	JPRO
program size (lines)	18386	12654	7199
classes with method tables	24	11	12
classes without method tables (e.g. abstract)	13	10	3
total number of method applobjs	5569	667	330
total number of methods invoked (typical)	750(13%)	218(33%)	30(9%)
number of `defmethods` in the program text	453	154	204
applobjs created by inheritances	5516	513	126

and TCP/IP networking system NET [9] are used.

Table 3 shows the data obtained from these applications at the time of this paper was being written. Even now, these programs are used practically and are being improved. The table 3 shows that only 10 to 30% of all `defmethod`'ed methods are actually invoked. This proves the effectiveness of the on-demand method construction. Such low utilization of methods is likely to be caused by the inheritance, which incorporates a lot of methods from superclasses even if most of them are never used. The on-demand applobj construction reduces the memory consumption caused by unnecessary inheritance. On the other hand it suggests that a new scheme providing a flexible inheritance control mechanism is desirable.

ZEN and NET utilize the inheritance mechanism heavily. However, they do not use method combination very often. (Data is omitted here.) Even though TAO provides a wide variety of method combinations, ZEN and NET use simple ones like the `:daemon` combination. The main purpose of the inheritance mechanism is modular programming, which works effectively in those applications. But it seems that the method combination is not so practical to use.

5. Conclusion

TAO object oriented programming can be said to be successful from the performance viewpoint. There are three dominant factors that determine the performance: (i) recognition of message passing forms in the interpreter, (ii) method search and its invocation, and (iii) execution of the method body. In TAO, (i) a message passing form is recognized in the efficiently microcoded `eval`, and (ii) the method search achieves a practical performance with a microcoded binary search algorithm. (iii) As the method body execution is the same as that of a Lisp function, there is no additional overhead. Instance variable access is quite fast thanks to a hash technique. As a result, TAO's object-oriented programs in the interpreter run fast enough in comparison with Lisp programs. In addition, by the on-demand method for constructing internal structures for object-oriented programming, the memory consumption and turn around time during program development are greatly reduced.

However, there are lots of issues left in object-oriented programming itself. Among them is that the inheritance and the complicated method combination scheme make the program's understandability quite low. Although information hiding is accomplished between objects, that between classes in the inheritance relationship is hardly achieved. If a programmer wants to use the already existing classes as his/her programming parts, he/she must know the implementation of the classes in detail. Superclasses cannot be seen as black boxes. It imposes an additional burden on the programmer. Furthermore, the object-oriented programming system does not provide any guidance for modeling, i.e., the system does not say what should be objects. It would be desirable for the system to give this kind of guidance, if it aims to be an intelligent programming system. Those issues, which request various concepts to be reviewed and a number of new ideas to be introduced, are still open to further investigation.

Acknowledgments

The authors are very thankful to those who support this work by using the system, reporting bugs, giving suggestions for improvements, and providing valuable comments. Among them are Hiroshi G. Okuno, the co-implementor of TAO kernel, Yasushi Hibino and Kazufumi Watanabe, the ELIS hardware designers, Yoshiji Amagai, who wrote ZEN, Ken'ichiro Murakami, the author of NET, Minoru Kamio, the implementor of TAO compiler, and many many other people both inside and outside NTT, including NTT Intelligent Technology Co. and Oki Electric Industry Co., Ltd. who are the co-developers of commercial versions of the TAO/ELIS system.

References

1. HIBINO, Y., WATANABE, K. and OSATO, N. The architecture of the Lisp machine ELIS—the memory-general registers and their effects—. *IPSJ SIG Note*, **24**, 3 (July 1983) (in Japanese).
2. TAKEUCHI, I., OKUNO, H. G. and OSATO, N. TAO—A harmonic mean of Lisp, Prolog and Smalltalk. *ACM Sigplan Notices*, **18**, 7 (July 1983), 65–74.
3. OKUNO, H. G., TAKEUCHI, I., OSATO, N., HIBINO, Y. and WATANABE, K. TAO: A Fast Interpreter-centered System on Lisp Machine ELIS. In *Conf. Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Tex., August 1984. ACM.
4. GOLDBERG, A. and ROBSON, D. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, Reading, Massachusetts, 1983.
5. WEINREB, D., MOON, D. and STALLMAN, R. M. *Lisp Machine Manual*. LMI, 1983.
6. SUZUKI, M. editor. *Object oriented programming: Tutorials and Reports from WOOC'85*, Kyoritsu Publishing Co. (1985) (in Japanese), 105–118.
7. KAMIO, M. A Common Lisp compiler on ELIS. *Proc. the 34th Conference of IPS*, number 1P-4, 1987 (in Japanese).
8. AMAGAI, Y. Making programming parts of a display editor based on object oriented programming. *Proc. the 33th Conference of IPS*. (1986) (in Japanese), 771–772.
9. MURAKAMI, K. An object oriented implementation of LAN protocols, *Proc. the 34th Conference of IPS*, number 1P-7 (1987) (in Japanese).