TAO : A Fast Interpreter-Centered System on Lisp Machine ELIS

Hiroshi G. OKUNO, Ikuo TAKEUCHI, Nobuyasu OSATO, Yasushi HIBINO and Kazufumi WATANABE

Musashino Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation
3-9-11 Midoricho, Musashino, Tokyo, 180 Japan

"The Tao that can be Taoed is not the true Tao."
Lao-Tse

ABSTRACT

This paper describes the design issue, implementation and performance of a Lisp called TAO for the Lisp machine ELIS. TAO is a dialect of Lisp which unifies an object-oriented programming paradigm and a logic programming paradigm with a procedural programming paradigm. Since the interpreter is implemented fully by microcode, interpreted code runs faster than most of other dedicated machines.

1. Introduction

A dialect of Lisp, TAO, is designed as a kernel language for an intelligent total programming system called NUE (New Unified Environment). Although TAO supports most of features of Zetalisp [WEINREB 83], it is not a mere dialect of Lisp, but also assimilates the essence of object-oriented programming and logic programming into Lisp world.

On NUE, AI researchers will be able to develop large scale programs in a highly interactive manner via advanced audio-visual terminal. NUE will be implemented on a Lisp machine ELIS developed by us [Hibino 83].

The design and implementation philosophy is that a wide variety of functions should be provided with fast execution speed. In the interactive environment, it is essential for interpreted code to run as fast as possible.

Since basic computational mechanisms for Lisp, Smalltalk and Prolog are fused in a sound manner and implemented directly by microcode, interpreted code runs faster than or at least as fast as other single paradigm dedicated machines. To implement a new Lisp system, Lisp-in-Lisp approach is often adopted [Brooks 83]. This approach gives high portability but slow interpreter which should be compensated by compiler. That is not what we want.

In Chapter 2, Lisp machine ELIS is described. In Chapter 3, the design philosophy and features of TAO are discussed. In Chapter 4, some topics on TAO implementation on ELIS is discussed and in Chapter 5, the performance of TAO is profiled.

2. Lisp machine ELIS

The Lisp machine on which TAO is implemented is called ELIS. ELIS is designed and manufactured by ourselves. The specs and outstanding features of ELIS are summarized below.

- 32/24 bit two mode ALU
- 32 bit Lisp pointer with 8 bit tag included
  Tag specifies what the pointed object is, not what it is.
- 24 bit addressing space of 64 bit cell (16M cells at maximum, or 128M bytes)
- Real memory system, not virtual memory (It will be feasible in the future.)
- Microcycle time is 180ns and memory access time is 420ns
  Three non-memory microinstructions can be performed behind one memory cycle.
- 32K 64 bit writable control store (planned to extend to 64K)
- 32K 32 bit stack with three stack pointers
  Stack push-down and pop-up operations are both performed in one microcycle.
- Four 64 bit memory general registers used as memory data registers or address registers (called MGR)
  Data is read from memory to a MGR in 64 bit width and written to memory in 64 bit or 32 bit width. Car part or cdr part of these registers can be used as general registers for ALU operations. MGR alleviates the memory operation neck of list processing to a

certain extent.
- Three index registers that point to byte positions of MGR, useful for string manipulation and LAP (compiled) code interpretation
- Hardware check of memory access by a special tag bit which shows the car-cdr accessibility of a Lisp object
- Simple and repetitive architecture (16K gates) suitable to one chip VLSI

ELIS is not a machine specially tuned for LISP but rather a general purpose machine good at memory accessing and stack operations. Hence, software architecture plays a very important role to make the best of it.

3. TAO Language Design

Lisp is continuously evolving in order to absorb or express new concepts as natural languages are. We think that this eventual evolution fascinates researchers on AI and makes them adopt Lisp as programming tool. The philosophy for designing TAO on ELIS stresses sound balance between richness of functionality, implementation ease and performance.

> All that and only that is judged useful for practical programming and can be implemented smoothly and efficiently on ELIS firmware is incorporated into TAO.
>
> Richness of language semantics and how well to use it (programming discipline and style) should be separate issues. That is, languages should provide as large and powerful (maybe dangerous in a sense) facilities as possible with the best performance.

Lisp is considered as a language based on lambda calculus and s-expressions. But we think that the power of Lisp comes from s-expression, a simple but versatile data structure. It could accommodate more computational semantics, which has been proved by a number of researchers who experimented new computation models on Lisp.

TAO provides s-expression with lambda calculus, restricted predicate calculus, message passing, data abstraction, and conventional von Neumann type computation mechanism. In other words, s-expression accommodates Lisp, Prolog, Smalltalk, and Fortran semantics in a harmonic way. Users can program by selecting and mixing the following programming paradigms:

- Procedural, like Lisp and Fortran
- Object-oriented, like Smalltalk
- Logic, like Prolog

By "harmonic", it is meant that these programming paradigms are fused in "eval".

3.1 Features in Lisp

TAO is almost compatible with major Lisps (especially MacLisp's family). TAO supports multi-user multi-process and real time (not parallel and not incremental) garbage collection. But they are now under construction. Details are omitted here.

(1) Data types

TAO supports a wide variety of data types like major Lisps. Six bits in the tag field are used for primitive data types, some of which are summarized in Appendix. As can be seen, Jstring covers Japanese text processing, bignum and rational cover exact mathematics such as symbolic algebra, and a set of locative data types covers Fortran-like number crunching. Especially, if locatives and message passing forms are combined, it is quite direct to write Fortran-like programs. If a variable is declared as a locative, an address to 64 bit word is assigned to it. For example,

```
(signed-integer-locatives x y)
```

declares variables x and y as a signed integer locative. The "dereferenced" values of x and y are 64 bit signed integer. Then, one can write such as:

```
(x := (x + (2 * y)))  .
```

(2) Multiple name space

Though the uniqueness of identifiers is one of the most outstanding features of Lisp, the larger the program grows, the more often identifiers come to conflict with each other; different functions come to have the same name. This uniqueness of identifiers prevents the modularity of programs and data. Lisp, as a total programming system, has to provide a multiple name space, as most operating systems have.

TAO has a feature called "oblist graph". The oblist graph differs from the package in ZetaLisp in that, besides the static tree structure of individual oblists, the user can specify a dynamic access path for his convenience. That is, static hierarchy of oblists corresponds to a hierarchical structure of directory in a file system, while dynamic structure corresponds to a logical name by which files should be sought. For example, "sys!foo" is an atom created in the oblist "sys", and "!rest" is an atom created in the "keyword" oblist. When interning a string, the atom of the same print name is sought in the accessible oblists. If such an atom is found, it is the value of intern. Otherwise new atom is interned in the specified oblist.

(3) Variable scope

A Lisp function (or resolver, which is described later) is either scope limiting or scope transparent. In other words, TAO supports static scoping and dynamic scoping. A scope limiting function makes a barrier for local variable access; that is, it puts up a barrier beyond which only variables explicitly declared "special" are accessible. On the contrary, a scope transparent function does not put up such a barrier. Hence, if all functions are scope transparent, all variables are dynamically bound like MacLisp. Scope limiting functions are defined by "de" and scope transparent functions by "dye" or "defun". Appropriate mixing of these would enhance the program modularity.

(4) Assignment

Assignment in TAO is more general and powerful than other assignment devices such as Zetalisp's "setf". Assignment to Lisp variables, arrays and lists (in the sense of rplaca and rplacd) are all representable in a simple, standard format: "(:assignee value)". The following examples show the power of TAO's assignment.

```
(:x v) = (setq x v)  if x is a variable.
(:(tbl i) v) replaces the i-th element
  of an array tbl by v.
(:(caddr x) v) = (rplaca (cddr x) v).
(:(third x) v) = (rplaca (cddr x) v),
  where "third" is defined as
  (de third (x) (caddr x)).
(:(member x l) v)  replaces the first
  element of l which is equal to x by v.
```

This assignment form is implemented by microcode, not by macro, unlike MacLisp's "setf". The key point is that a set of special internal registers is used to remember the assignable "position" which is most recently accessed by the user. These registers are updated every time the user access to assignable Lisp object via certain functions in the course of assignee evaluation. (Quite luckily, this does not introduce any overhead because of parallel microinstruction executions behind the memory access.)

Moreover, TAO provides the facility to eliminate double evaluation of an assignee position if the assigned value also contains the position. This form is called "self-assignment" and represented by "(::" with assignee position prefixed by ":". For example,

```
(::cdr :queue) = (:queue (cdr queue)),
(::f x (g (h :y) z)) =
       (:y (f x (g (h y) z))),
```

unless the assignee evaluation has no side effect. The background mechanism is almost the same as that of assignment.

(5) S-expression as a surface language

Many people prefer the conventional functional notation, such as fn(x), to Lisp's (fn x). TAO allows the user to use functional notation in I/O. If the variable "fn-notation" is non-nil and an identifier is input immediately (without blank) followed by an open parenthesis (or bracket), the identifier is shifted down into car of the succeeding list and the first cell of the list is tagged as named cell. Thus, "fn(x)" in functional notation has the same meaning as "(fn x)". However, it can still be output as "fn(x)". Using this option together with the message passing yields a natural representation of algebraic expressions, such as (f(x) + g(y)), which noticeably needs no one-way, irreversible pre-processing.

Similarly, tags are used extensively to fill a gap between external and internal representation of s-expression and to make the external one more readable. For example, "()" and "nil" are discriminated with respect to I/O.

Quoted objects are treated as a single Lisp object. Therefore, "(bar . 'foo)" and "(bar quote foo)" are quite different objects. The cadr of "(bar quote foo)" is "quote", while that of "(bar . 'foo)" is "foo". To make a quoted object, the function "quotify" must be used. Backquoted object "`(fn x)" and hat object "^(a b `c)" have similar structures.

Any number input in the octal radix, such as "#3323", is output in the same format in spite of the current base, while #3323 is equal to its decimal counterpart 1747 in numerical comparison.

3.2 Object-oriented programming in TAO

Object-oriented programming in TAO provides the mechanism for data encapsulation (information hiding) and data abstraction. TAO attempts to union the features of Smalltalk-80 and Flavor, since each of them has its own useful features which the other lacks.

(1) Primitives and user defined objects

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

Udo is an instance of some class which categorizes objects of the same kind. A class can be defined in the following form:

```
(defclass class-name
          class-variable-list
          instance-variable-list
          !opt superclass-list
          !rest defclass-options ) .
```

The class definition is stored in the property list of the class-name. Class-variable-list and instance-variable-list declare a set of class variables and instance variables. The optional superclass-list specifies a set of superclasses. If it contains plural superclass names, the class inherits all the instance variables and methods of the superclasses unless they are intentionally overridden. This mechanism is called "multiple inheritance".

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 by traversing the network in depth-first left-to-right order. Inherited methods are registered to the class, picked up by the same network traversing.

The !rest (so-called nospreading lambda) argument may contain various optional specifications of this class. It may specify default creation of some kinds of methods such as those to get the value of a variable or to set a value of a variable. It may also specify a special action to do when an instance is created.

The user can add new methods to system defined objects, and can override or mix methods except for the system defined messages. For example, Fibonacci function can be defined as a method on the primitive class integer (Figure 1-(b)).

TAO has no notion of metaclass. Classes need not be objects. Methods which correspond to class methods in Smalltalk-80 are defined as Lisp functions. For example, instance is created by "make-instance" and method is defined by "defmethod".

(2) Message passing form in TAO

Message passing is realized in genuine form like Smalltalk-80, not a function call. 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 an applicative object, e.g., function, array, macro, or etc. Detecting that a form is message passing does not bring in overhead to usual Lisp form evaluation, since it just fills up an error reporting slot in conventional "eval". 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 a special tag. Only "eval" switches its action on the tag. Bracketed form is essential when the receiver itself is an applicative object. Suppose that an identifier "list" has a udo as value. In this case, "(list foo bar)" means a simple function application, while "[list foo bar]" means a message passing. It also serves as a declaration for compiler optimization.

Message-pattern is either an identifier or a list, and it is not evaluated unless it is backquoted. 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.

(3) Infix notation

Message passing form in TAO enables the user to express an ordinary arithmetic infix notation like "(A + B)". Infix notation together with function notation enhances the program readability. In addition, nested message passing forms can be flattened into a single list for some system defined messages, if the first form is the receiver of the second form and the second is the receiver of the third and so forth; e.g.,

```
(x + 4 - y + (x * z) + u) .
```

(4) Type polymorphism

Generic operation can be defined for various data types. Since the code for the operation is defined separately for each class, the data type check is not necessary in the body of the operation. For example, a generic operation ".." invokes append to a list and string-append to a string, e.g.,

```
((list 1 2 3) .. '(4 5)) ==> (1 2 3 4 5)
("con" .. "cat") ==> "concat" .
```

(5) Method combination

In TAO, methods of the same message name can be combined statically like Flavor. Hence, the fuss of combining methods in the specified way is done only once. A combined method is itself an independent method and is registered to the class.

(6) Super's method invocation

To access directly to a particular superclass's method, a compound message is used. For example, a compound message "foo.initialize" invokes a method corresponding to the message "initialize" in a superclass "foo". "foo.initialize" is itself a single identifier and becomes the message name defined in this class (not "foo"'s). 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. The method is sought in the class's table. If "class.message" is not found in this table, then the compound identifier is decomposed and the corresponding class's table is searched. If the desired method is found, it is redefined in this class and added to the class's table.

Another form:

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

invokes a particular overridden method in some superclass. This form is just the counterpart of Smalltalk-80's.

Similarly 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 ...)
```

and list-message which causes the method search start from the direct superclass:

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

3.3 Logic programming in TAO

Logic programming will be useful in some programming areas such as database manipulation and lexical analyzing. From the viewpoint of operational semantics, there are two essential functions in Prolog: unification and backtracking. Unification is functionally more powerful than simple pattern matchings proposed in various Lisp dialects, since a pattern which is only partially instantiated can be used in the course of the computation. These two essential functions are introduced separately in TAO. Their efficiency is, of course, maximally guaranteed. Hence, the user can combine these functions to make up his own Prolog(-like) system without loss of efficiency and generality.

(1) Unification

In TAO, a unification takes place as a function application to a pattern to be unified. On the other hand, in Prolog, it is invoked automatically by the existence of a negative literal. An applicative function which causes unification is called a "resolver". The resolver resembles the Lisp's lambda in the sense that they are both the basic applicative function types (applobj). Note that both resolver and lambda are not a function per se, but that they will create a function object when they are evaluated. That is, any s-expression in its written form does not work as a function as in conventional Lisps. Function object is an independent data type which should be created by some computation.

There are two types of variables in TAO; one for Lisp and the other for logic. Logic variable is introduced to speed up variable check in unification. Since they are quite different data types internally, logic variables is distinguished from Lisp variables by a user customizable syntax. For example, logic variable is prefixed by "*" or begins with a capital letter. (In this paper, the former syntax is used.)

A Horn clause with one positive literal:

```
A :- B1, B2, ..., Bn.
```

which reads "to execute A is to execute B1, B2, ..., Bn in this order" in the procedural interpretation. This is represented in TAO as follows:

```
(&+ A [(!aux var ...)] B1 B2 ...  Bn) ,
```

which creates an "&+ resolver".

The &+ resolver tries to unify its arguments with A, the header of the resolver. If the unification fails, the value of the application will be nil. If it succeeds, B1, B2, ..., Bn, the body of &+ is evaluated sequentially. In fact, the body is not a resolver, but a logic counterpart of Lisp's "prog", or more precisely Lisp's "and". First, &+ body declares the (optional) auxiliary variables and then evaluates Bi's in sequence. If all the values of Bi's are non-nil, the value of &+ body is the value of the last literal Bn. (If the body is nil, t is returned.) If some Bi returns nil, it is interpreted as failure. If failure occurs, the execution of body is backtracked to the most recent choice. The computation continues from the backtracked point with next alternative, after unwinding some effects that have caused the failure. If all the alternatives exhaust the value of the &+ body will be nil.

The following simple examples show the contrast between lambda and resolver.

```
((lambda (x) (car x)) '(1 2 3))

((&+ ((*x . *y)) *x) (1 2 3))
```

The first binds x to "(1 2 3)" and returns 1, the value of "(car x)". The second

instantiates *x to 1 and *y to "(2 3)" respectively, and returns 1, the value of *x. Note that arguments of a resolver are not evaluated in Lisp sense.

Part of the argument, however, may be evaluated before unification, if it is prefixed by a backquote symbol "`". For example, "(&append *x *y `foobar)" will be "(&append *x *y (1 2 3 4))" eventually if the value of the Lisp variable foobar is "(1 2 3 4)". Backquote notation allows the user to avoid a clumsy "load-and-store assembly programming" in Prolog. For example, factorial is defined in Prolog,

```
fact(0,1).
fact(N,V) :- N1 is N-1, fact(N1,V1),
             V is N*V1.
```

while in TAO,

```
(assertz (fact 0 1))
(assertz (fact *n *v)
         (fact `(*n - 1) *v1)
         (== *v `(*n * *v1)) ) .
```

If any sub-pattern in the &+ definition is quoted, it matches only fully instantiated object. For example,

```
((&+ ('Turing)) Turing) ==> t

((&+ ('Turing)) *y) ==> nil
```

while

```
((&+ (Turing)) *y) ==> t
```

where *y is supposed to be declared somewhere but uninstantiated yet.

(2) Backtracking

Backtracking is realized by another resolver type, Hclauses, which corresponds to a set of Horn clauses:

```
(Hclauses
  (&+ A [(!aux var ...)] B1 B2 ... Bn)
  (&+ A' [(!aux var' ...)] B1' B2'... Bm')
  ... ) .
```

This form returns an Hclauses resolver, namely a set of &+ resolvers.

When an Hclauses resolver is applied to its arguments, each &+ resolver is applied to the arguments in sequence until one of them returns non-nil value. However, it differs from a Lisp's "or" in that it preserves untried alternatives to resume computation on later backtracking. For example, the following form returns the value 1 at first.

```
((Hclauses (&+ (*x *y . *z) *x)
           (&+ (*x *y *x) *y))
 1 (*a 2) *a )
```

However, the next alternatives are all preserved, if this form is evaluated in an (explicit or implicit) &+ body. If a later computation within the backtracking scope leads to failure, the next alternative is chosen. In the above example, the form's next value is "(1 2)". One more backtracking exhausts the alternatives. Hence, the final value of the form will be nil, that is, the "literal" turns out to be false. Note that &+ concern only unification while Hclauses concern only backtracking.

(3) Logic variables

Logic variables are declared explicitly in auxiliary variable declaration part of the &+ body or in "prog" variable declaration part, or implicitly in &+ resolver's header. Explicitly declared one is considered as a logic variable with existential quantifier, whose initial value is "undefined". Implicitly declared one is considered as a logic variable with universal quantifier, whose initial value is determined by the unification at the &+ resolver application.

The scope of logic variables is the same as that of Lisp variables. That is, there are two types of resolvers with respect to the variable scope: scope limiting and scope transparent. Scope transparent &+ resolver is created by "&+dyn".

(4) Backtracking scope

When a failure happens in an (explicit or implicit) &+ body, i.e., an element of the &+ body returns nil, a backtracking occurs. In that case, the instantiation effects that have taken place since the most recent choice of alternatives (the most recent Hclauses resolver application) are unwound, then the next alternative is chosen and the computation proceeds just from the point of the choice again. However, there are some subtle points about backtracking as follows:

(i) Lisp form is not backtracked and any effect that has taken place within the Lisp form is not unwound. Even the instantiations in a resolver application which is nested in a Lisp form is not unwound. In other words, preserved alternatives are all discarded when value is returned into a Lisp environment.

(ii) If a cut symbol "!" appears in an &+ body, the alternatives of the innermost goal statement ! or Hclauses which is lexically visible in the scope are discarded. If there is none, the cut symbol yields no operation.

(5) Naming resolvers

As a lambda can be associated with an identifier by "defining a function" in Lisp, any resolver can be associated with an identifier. In a primitive form, for

example, one can write:

```
(defrel == ((*x *x)))

(defrel &append
   (((*a . *x) *y (*a . *z))
    (&append *x *y *z) )
   ((() *y *y)) )
```

or,

```
(assert (== *x *x))

(assert (&append (*a . *x) *y (*a . *z))
        (&append *x *y *z) )
(assert (&append () *y *y))
```

Auxiliary variable declaration is not necessary in an assert form, because assert creates it automatically. An identifier associated with a resolver is just what is called a functor in Prolog.

(6) Goal statements

A goal statement is a Horn clause with no positive literal (in Prolog notation):

```
?- B1, B2, ..., Bn.
```

In the procedural interpretation, this reads "execute B1, B2, ..., Bn in this order". This clause is represented as follows:

```
(& [(!aux var ...)] B1 B2 ... Bn) .
```

A goal statement is considered as a variant of &+. That is, an "&" form is an &+ application without header unification. In fact, "&" form is defined by using macro as follows:

```
(defmacro & (!rest body)
  `((&+dyn () `@body !)) )
```

where "!" means "cut" in Prolog; "`" and "`" are the counterpart of "`" and "," in MacLisp's backquote macro facility. That is, this form is equivalent to:

```
(defmacro & (!rest body)
  (list (append '(&+dyn ()) body '(!))) ).
```

An "&" form "(& B)", which has only one negative literal and no auxiliary variable declaration, can be written in a shorter form, simply B. This form is quite natural and convenient when mixed in Lisp forms as shown below. Assume that there is a database written in the Prolog assertion form on humanity as:

```
(assert (human Turing)),
(assert (human Socrates)), ... .

(de Supergirl (x)
  (cond ((and (human `x) (friendly-p x))
         (fist nil)
         (shake-hand-with x) )
        (t (fist t) (strike x)) ))  .
```

"&" form corresponds to logical "and" in the sense that logic involves automatic backtracking. The following "!" form and "~" form correspond to Lisp's "or" and "not", respectively.

```
(! [(!aux var ...)] B1 B2 ... Bn)
```

evaluates B1, B2, ..., Bn in this order. It returns the first non-nil value of Bi. "!" is almost the same as Lisp's "or," except that it has optional variable declaration and preserves the backtracking scope.

```
(~ [(!aux var ...)] B)
```

evaluates B and returns its negation, i.e., t if B's value is nil, nil otherwise. "~" is almost the same as Lisp's "not", except that it has optional variable declaration. "~" is implemented as negation by failure in the logic programming terminology. Note that the forms "(&)" and "(!)" evaluate to t and nil, respectively.

4. Implementation topics

4.1 Invisible pointers

Invisible pointers which the user cannot see in usual I/O and by primitive functions such as car and cdr are extensively utilized to speed up the interpreter, though they are originally introduced to implement unification in TAO (4.6).

TAO adopts deep-binding scheme in order to alleviate the burden of process switching. It is, however, more expensive in this scheme for the interpreter to access variables than in shallow-binding scheme. This problem is solved by inserting invisible pointers which associate each variable (Lisp, logic, and instance variables) with its relative position in the activation frame. Invisible pointer insertion is performed at the time of function definition. The user cannot notice that such information is embedded in his program, but his program runs about 30 to 50 percent faster by this technique.

Invisible pointers are also utilized for storing expanded macro/subst expression, branch table for selectq (sorted), go-label table in prog body and such information that is once "semi-compiled" in order to speed up the interpreter without destroying original program (at least from the user's viewpoint).

Other usages of invisible pointers are multi-value, growlist (list that can grow at its end by tconc), and comment.

4.2 String representation

Japanese string (or Jstring, for short)

consists of two 8 bit strings: upper string and lower string, because Japanese code is 14 (7 + 7) bit wide and stored in two bytes. String with font (so-called fat-string) and Jstring with font have similar structure. Hence, string manipulating functions can share common routines and the compatibility between alphanumeric strings and Japanese string can be easily achieved by microcode.

4.3 Method lookup

Each element of id-message table is a pair of id-message name and corresponding method which is defined in the class or inherited from its superclass. The pairs are sorted by the address of message name so that methods can be looked up by binary search strategy. In this table, methods to get value of an instance variable or set the value are represented by integers (positive for getting, negative for setting) which denote the relative position of the instance variable in the instance variable table.

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.

4.4 Lazy principle

Various runtime information on udo's class is not created before it becomes necessary. This is because the system allows incremental definition of classes and methods. For example, skeleton for instance creation is constructed at the time of first instantiation of this class; id-message table is not created before the first id-message is sent to some instance of this class.

Id-message table 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.

4.5 System defined messages

Primitive operations on system defined objects represented by system identifier such as +, -, .(dot), ..(double dots) etc., are processed in quite an efficient manner. 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 microroutine and the control is transferred to it. Similar technique is also used to speed up the Lisp primitives such as car, cons and eq.

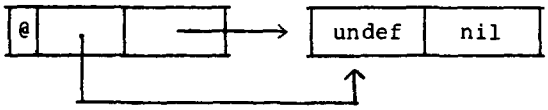
Since all primitive objects do not take udo structure, class descriptions of these objects are maintained in fixed system data area.

4.6 Structure copying unification

Unification is implemented by copying, not by structure sharing, because logic world and Lisp world should have common data structures. The shared structure is represented by the aid of two kinds of invisible pointers: invisible-car and invisible-cdr. For example, suppose that *x and *y are not instantiated, then unifying *x and (*y *y), or executing the following form:

```
(== *x (*y *y))
```

makes the value of *x as follows:



The car of first cell is an invisible-car pointer, which indicates that the ultimate value is the car of the cell whose address is specified by the invisible pointer. By introducing invisible pointers, certain overhead is expected to be brought in to Lisp primitives such as car, cdr, cons and eq, but it is not the case. Since the invisibility check is absorbed in usual data type check for these primitives, no timing loss is brought in.

5. Performance Evaluation

The first goal of implementation is a fast interpreter which enables programmers to select any paradigm without bothering about the speed difference between them. The goal is paraphrased as follows:

> The performance of three paradigms (Lisp, Prolog, and Smalltalk) should be well-balanced.
>
> $1 < \text{Lisp, Prolog, Smalltalk} < 1.5$

The success of this goal depends on how extensively to make use of ELIS hardware architecture. In this chapter, some timing results are given.

5.1 Lisp programming

The results of various benchmarks are shown in Table 1. These benchmarks are those used in the 2nd Lisp contest in Japan [Takeuchi 76]. The results on TPU (Theorem Prover by Unit binary resolution [Chang 73], about 400 lines) may reflect the speed of daily jobs.

	TAO/ELIS interpreter	MacLisp/Dec-2060 interpreter	MacLisp/Dec-2060 compiler
sort-80	390	3,255	145
sort-100	540	3,980	361
tak(10,5,0)	16,900	125,900	4,390
tak(12,6,0)	621,000	5,241,000	157,000
TPU-3	900	5,242	1,142
TPU-6	4,580	27,569	4,757
TPU-9	560	3,182	343
		(unit: millisecond)	

Table 1 Comparison between TAO and MacLisp

5.2 Object-oriented programming

We have no common benchmarks on object-oriented programming languages. Hence, the speed in object-oriented programming is compared with that in Lisp. The definitions of benchmarks are shown in Fig. 1 and its execution time in Table 2. In TAO, user-defined methods are sought by binary search at the execution time while the Flavor adopts hashing. In Table 2, various situation of user-defined methods on the class integer are compared:

(i) There is only one user-defined method, fib
(ii) There are thirty user-defined methods including fib and fib is found in the worst case
(iii) There are one hundred user-defined methods including fib and fib is found in the worst case.

(a) Lisp style Fibonacci

```
(de fib (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2)) ))))
```

(b) Object-oriented style Fibonacci

```
(defmethod (integer fib) ()
  (cond ((self = 0) 1)
        ((self = 1) 1)
        (t (((self - 1) fib) +
            ((self - 2) fib) ))))
```

Fig. 1 benchmarks for object-oriented programming

	Procedural (fib n)	Object-oriented (n fib)		
		(i)	(ii)	(iii)
fib-19	742	798	840	872
fib-22	3,142	3,386	3,570	3,694
fib-25	13,322	14,334	15,110	15,664
		(unit: millisecond)		

Table 2. Timing results

It is shown that the execution speed of object-oriented programming is only by 7 to 15 percent slower than that of Lisp.

It takes about 10 microseconds to access to an instance variable of a user-defined object by the message of the same name of the (so-called gettable) variable.

5.3 Logic programming

We adopts a list reverse function as a benchmark, shown in Fig. 2, because its number of LI (Logical Inferences) becomes widely accepted now. It is said that it takes 496 LI to reverse a list of thirty elements. The execution time of logic programming and its corresponding Lisp programs is shown in Table 3.

(a) Lisp style reverse and append

```
(de append (x y)
  (cond ((consp x)
         (cons (car x)
               (append (cdr x) y) ))
        (t y) ))

(de reverse (x)
  (cond ((consp x)
         (append (reverse (cdr x))
                 (list (car x)) ))
        (t nil) ))
```

(b) Prolog style reverse and append
(See 3.3.5 for &append.)

```
(defrel &reverse
  (((*x . *y0) *y)
    (!aux *y1)
    (&reverse *y0 *y1)
    (&append *y1 (*x) *y) )
  ((nil nil)) )
```

or,

```
(assertz (&reverse (*x . *y0) *y)
         (&reverse *y0 *y1)
         (&append *y1 (*x) *y) )
(assertz (&reverse nil nil))
```

Fig. 3. Benchmark for Lisp and Logic

Procedural programming	23.3 ms
Logic programming	44.0 ms

Table 3. Execution timing of reverse

From this example, it is possible to say that TAO's power of logic programming is over 11K LIPS in interpreter. Logic programming in TAO is, however, nearly two times slower than Lisp programming. This slow down of speed may be compensated by the bidirectionality of logic computation. Consider the following tricky program which Lisp by no means comes up with:

```
(prog (*1 *2 *3 *4 *5 *6)
      (&reverse (*1 2 *3 4 *5 6)
                (*6 5 *4 3 *2 1)  )
      (return (list *1 *2 *3 *4 *5 *6)) ).
```

6. Concluding Remarks

TAO combines procedural, object-oriented, and logic programming paradigms in a harmonic way. As for Lisp, TAO is almost compatible with major Lisps. TAO is also semantically compatible with DEC-10 Prolog and Smalltalk-80. (Indeed, DEC-10 Prolog surface syntax will be supported.) We believe that Lisp vs. Prolog controversy is somewhat meaningless once they are combined in this way, and that object-oriented programming is more important to improve our programming. Logic Programming is one, but not all, of the useful paradigms that next generation programming systems should have.

The execution speed ratio of procedural programming to object-oriented programming is shown to be less than 1.2, and the ratio of procedural programming to logic programming is less than 1.9. We think that the principal goal of TAO implementation is almost achieved. This enables the user to choose and combine paradigms adequate for the desired purposes without tuning their programming style to the performance profile.

One of the currently running non-toy applications on TAO/ELIS are the micro-assembler and micro-linker for ELIS which were initially implemented in a small but fast Lisp on PDP-11. The new system on ELIS assembles and links microprograms about four times faster than the old one which was executed in compiled code.

Batch type garbage collector is now running and it will soon become real time one when multi-process kernel in microcode is accomplished. Compiler is now under development to run code three to five times faster than that of interpreter. This year, more than five ELIS's are going to be made and used as software tools for AI research at our laboratories.

Acknowledgments

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

References

[Brooks 83] R.A. Brooks, R.P. Gabriel and G.L. Steele Jr. Lisp-in-Lisp : High Performance and Portability, Proc. of 8th IJCAI, Aug. 1983.

[Pitman 83] K. M. Pitman. The Revised MacLisp Manual, MIT AI Lab., May 1983.

[Warren 77] D.H.D. Warren. Implementing Prolog - compiling predicate logic programs, D.A.I. Research Report, 39 and 40, Univ. of Edinbourgh, May 1977.

[Weinreb 83] D. Weinreb, D. Moon, and R. Stallman. Lisp Machine Manual, Fifth Edition, LMI, Jan. 1983.

[Goldberg 83] A. Goldberg and D. Robson. Smalltalk-80: The Language and its implementation, Reading, Massachusetts, Addison-Wesley, 1983.

[Takeuchi 83] I. Takeuchi, H. G. Okuno, and N. Osato. TAO - A harmonic mean of Lisp, Prolog and Smalltalk, SIGPLAN NOTICES, Vol.18, No.7, July 1983.

[Hibino 83] Y. Hibino, K. Watanabe, and N. Osato. The architecture of Lisp machine ELIS (in Japanese), Report of SIGSYM, 24, IPSJ, June 1983.

[Takeuchi 78] I. Takeuchi. Report on Lisp Contest, Report of SIGSYM 5-3, IPSJ, Aug. 1978.

[Osato 83] N. Osato, H. G. Okuno, and I. Takeuchi. Object-Oriented Programming in Lisp, Report of SIGSYM, 26-4, IPSJ, Dec. 1983.

[Chang 73] C-L. Chang and R. C-T. Lee. Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

Appendix Data types of TAO (part)

undef	"value is undefined"
id	identifier
logic	logic variable
codnum	coded number with mnemonic I/O
shortnum	24 bit short number
bignum	arbitrary long integer
rational	rational number
float	64 bit floating number
bigfloat	arbitrary precision floating number
string	8 bit character string
fat-string	string with font information
Jstring	16 bit character string
fat-Jstring	Jstring with font information
vector	simple array of Lisp objects
applobj	applicative object (lambda, subst, macro, array, closure, resolver, etc.)
memblk	memory block (not necessarily composed of Lisp objects)
locbit	locative pointer into 1, 2, 4, 8, 16, 32 and 64 bit binary data memory block (movable)
64builoc	64 bit unsigned integer locative (immovable)
64siloc	64 bit signed integer locative (immovable)
64floc	64 bit floating locative (immovable)
udo	user defined object
cell	normal list
bracket	list used for message passing
growlist	list that can grow at its end
multivalue	multiple value