記号処理 37－7
（1986. 6. 20）

TAO/ELIS上でのCOMMON LISPの実現

竹内郁雄
日本電信電話株式会社 NTT電気通信研究所

COMMON LISPのフルセットを我々が開発中のTAO/ELISの上に実現した．このためTAO自身にいくつかの変更を施した．しかし、基本的にコンパイラ指向であるCOMMON LISPと，インタプリタ指向であるTAOでは，設計思想も基本構造も異なっている．本実現では，COMMON LISP特有の計算機構は，ほとんどTAOへS式レベルで変換する．ただし、この変換は"見えないポインタ"を使っており，ユーザにはCOMMON LISPのソースがそのまま解釈実行されているように見える．この手法により，実行速度はTAO自身とほとんど同じにすることができた．なお，TAOと相容れないためにCOMMON LISPパッケージに封入された関数は約50個で，COMMON LISP全体の約1/10である．

ON THE IMPLEMENTATION OF COMMON LISP ON TAO/ELIS

Ikuo Takeuchi

NTT Electrical Communication Laboratories, Nippon Telegraph and Telephone Corporation

The full set of Common Lisp is implemented on TAO/ELIS. TAO itself is changed to accommodate to Common Lisp to some extent. However, the most essential difference between them that Common Lisp is a compiler-centered language while TAO is an interpreter-centered language is resolved mainly by the "invisible" expansion of Common Lisp's forms into TAO's ones. That is, a simple "compilation" of Common Lisp into TAO at the S-expression level is done during defining functions, which is invisible from the users. The number of the functions which are excluded from the native TAO and packed into the Common Lisp package by a variety of reasons is only about 50. The performance of the Common Lisp on TAO/ELIS is comparable with that of the native TAO both in the interpreter and the compiler.

1. INTRODUCTION

The full set of Common Lisp language described in the Aluminum book [1] was implemented on TAO/ELIS. TAO is a Lisp dialect developed on a dedicated Lisp machine ELIS [2]. It is a multiple programming paradigm language combining the logic programming of Prolog and the object-oriented programming of Smalltalk-80 or Flavor with the conventional procedural and functional programming of Lisp. On the other hand, Common Lisp is one of the most promising standard Lisp languages. Many manufacturers announce to follow Common Lisp, and many users want it to be available on a variety of computers. Though TAO bears its own design principle and fundamental structure different from Common Lisp, it is desirable to support Common Lisp in TAO. Fortunately, "What is said the TAO is not the TAO" and "TAO is the source of everything." TAO is an ever evolving language. Implementing Common Lisp on (or in) TAO is one of the episodes of the evolution. This report will describe some technical topics of the Common Lisp implementation, mainly focusing on the interpreter. (The full compiler implementation will be presented in another paper.)

2. OVERVIEW

The Lisp part of TAO was originally designed to be upward compatible with MacLisp and ZetaLisp, and it bears many TAO specific features based on the principle that the language should run as fast as possible in the interpreter. However, the principle makes some features contradictory to or subtly different from those of Common Lisp. There may be two types of decisions to resolve the problem:

(1) Minimizing the change of TAO, and making a big Common Lisp package.
(2) Modifying TAO to be compatible with Common Lisp as far as permissible, and minimizing the size of the Common Lisp package.

We at first inclined to the decision (1), and then gradually changed to the decision (2) as we discovered devices and tricks by which the runtime performance could be made comparable with the TAO native mode. Among those mechanisms which might be expected to be implemented in the core the microcoded eval but actually are not are:

(1) Keyword arguments,
(2) Initial value for optional variables,
(3) Lexical scoping of block names and go tags,
(4) Automatic lambda closures,
(5) Local functions.

Instead, they are implemented almost by software, not by firmware, using some clever tricks as will be described below. Finally, the Common Lisp package includes some 70 functions. About half of them are string manipulation functions (e.g., *common:string-left-trim*) and input/output functions with keyword arguments, for which there are TAO native microcoded counterparts (e.g., *bas:string-left-trim*) with optional arguments instead of keyword ones. And these include some additional functions that are not presented in Aluminum book such as *common:nstring-left-trim*. Most of the rest are also packed into the Common Lisp package because their argument lists are more complicated than the TAO native microcoded (and, of course, faster) counterparts. They include *common:/=*, *common:logand*, and *common:apply*. (The last one may be controversial.) A few are functions for sequences: *common:length*, *common:reverse*, and *common:nreverse*. We decided that those functions of the same names for lists should be TAO native. (In most cases, for example, the function *length* is used to measure the length of a very short list, in which case the type discrimination done by *common:length* takes non-negligible time. The function *list-length* is not very fast because of the circularity check. TAO's native *length* detects a circularity by checking the counting overflow, which brings in no overhead for normal lists.) The last one is *common:lambda*, which will be described below.

Another solution of the name conflict of functions is renaming the functions which had been written in microcode, in a systematic way. For example, a family of membership functions consists of:

member	full Common Lisp version
mem	takes a test function as its first argument (Zetalisp)
memq	test function is *eq*
memql	test function is *eql*

memqu test function is *equal* .

That is, suffix "q" stands for *eq*, "ql" for *eql*, and "qu" for *equal*. Thus, very fast set operation functions such as *union* and *intersection* which utilize a hashing technique on "eq-able" objects in microcode are renamed to *unionq* and *intersectionq*, respectively. (We do not hide such functions behind the general functions of Common Lisp, since it is quite natural to give something frequently used its own name and that is the very history of the "natural" language.)

Functions in the Common Lisp package may be used by use-package all in one. However, it will be more reasonable for many users to select those they really need and *shadowing-import* them individually.

TAO had many queer notations originally. For example, assignment was represented formerly by *(:var value)*. It is now represented by *(!var value)*. Logical variable **var* is now represented by *_var*. We considered at first the syntax problem only as a matter of tuning the readtable. However, TAO finally followed the Common Lisp standard readtable convention in its native mode in order to avoid non-essential confusion. (This change imposed us much work than anticipated, since it set everything upside down in TAO which itself serves as the system development language for TAO and there was not the readtable yet.)

3. KEYWORD ARGUMENTS and OPTIONAL VARIABLES' INITIAL VALUES

TAO's native function to define a function is *de*. (There is another type of defining function *dye*, which defines a dynamic scoping function as MacLisp's *defun*. *De* is called scope limiting and *dye* is called scope transparent.) The following shows how optional variables are bound in *de*.

```
(de foo (&opt x y) (list x y))
(de bar (&optn x y) (list x y))
```

then,

```
(foo 1) -> (1 {undef}0)
(bar 1) -> (1 nil)
```

where *{undef}0* means a TAO specific data type denoting "undefined" which is introduced originally for the sake of logic programming.

It does not support keyword arguments, initial values of optional variables and supplied-p variables. To take in them in the firmware would cause the drastic change of the internal structure of TAO and make the system heavy and slow. Anyway, checking keyword arguments at runtime loads the system with a certain overhead.

The function *defun* (not enclosed in the Common Lisp package) will process a function definition as follows, if it contains features that *de* does not cope with:

(1) If the argument list specifies the initial value of an optional variable.

```
(defun foo (x &optional (y 1234) z) ...)
```

```
=> (de foo (x &optn y z) (funcall-init (!y 1234)) ...)
```

where *funcall-init* is an alias of *progn*. The alias is used in order to make it easy to get the original definition from the internal S-expression.

(2) If the argument list specifies a supplied-p variable.

```
(defun foo (x &optional (y 1234 ys) z) ...)
```

```
=> (de foo (x &opt y z)
       (funcall-init (progn (!ys t) (!!ifundef !y (!ys) 1234))
                     (!!ifundef !z nil) )
       ... )
```

where *(!!ifundef !var . forms) = (!var (ifundef var . forms))* and *(!var) = (!var nil)*. (The former is a self-assignment form peculiar to TAO.)

(3) If the argument list contains keyword arguments.

```
(defun foo (x &optional y &key :a (b 222) ((:c cc) y)) ...)

=> (defmacro foo (x &opt y &rest bas:$$r)
     (cond
      ; if it is applied or contains :allow-other-keys,
      ; check keyword arguments every time.
      ((or (and (quotedp bas:$$r)
                (!bas:$$r (image bas:i (car bas:$$r) (quotify bas:i))) )
           (get bas:$$r ':allow-other-keys) )
       `(apply 'foo-$$-internal
          (list* ,x ,y
            (bas:expand-key-args 'foo (list ,@bas:$$r) '(:a :b :c) 2) )))
      ; check the parity of keyword arguments
      ((not (evenp (length bas:$$r)))
       (error "odd number of keyword arguments" 'foo bas:$$r))
      ; replace the form by simpler one invisibly
      (t `(foo-$$-internal ,x ,y
            ,@(bas:expand-key-args 'foo bas:$$r '(:a :b :c) 2 nil) ))))

   (de foo-$$-internal (x &opt y a b cc)
     (funcall-init
      (!!ifundef !y nil)
      (!!ifundef !a nil)
      (!!ifundef !b 222)
      (!!ifundef !cc y))
     ... )
```

That is, *defun* yields one macro definition and one internal function definition. (Note that backquote, comma and such are not read-macro characters, but they indicate specially tagged cons data types like quote.) By this technique, typical function calling forms of foo are macroexpanded to very simple function calls as follows:

(foo qwe) to *(foo-$$-internal qwe {undef}0 {undef}0 {undef}0 {undef}0)*
(foo qwe asd :c 555) to *(foo-$$-internal qwe asd {undef}0 {undef}0 555)*

If a macro call form is once expanded, it will be never expanded again, hence no keyword argument check will be done at runtime more than once in these cases. And, of course, the compiler can easily generate an optimized code.

(4) If both rest and keyword arguments are specified. (omitted here)

A function with keyword arguments is applicable in spite of being defined as a macro by the following reason. TAO's macro can be applied or funcalled if it is so defined. User can declare a macro to be applicable in that case. The macro definition appeared in (3) above carefully makes it possible. (See the first conditional expression.) When a macro is applied, all arguments are attached with "quoted" tag; that is, if

```
(defmacro first (x) `(car ,x))
(funcall #'first (list 1 2 3))
```

then, macro body of *first* will get variable *x* bound to *'(1 2 3)*, not *(1 2 3)*. Hence, if the rest variable of a macro is bound to a quoted entity, the macro is detected surely to be applied! (It is still a problem that, when keyword parameters are supplied in a different order from that in the argument list and if one of them has a side effect which affects to other keyword parameter's value, the macroexpanded form would not work. But who knows?)

4. AUTOMATIC LAMBDA CLOSURE and LEXICAL GO/EXIT

We think that the function closure which explicitly designates the variables to be closed is still necessary to hold the binding control in hand, for example, in the case where the

closure will be used as a common communication channel for a number of processes. And we think also that the closed variables can be declared special by the same reason. This is the reason why TAO's *lambda* does not follow the Common Lisp's *lambda*. In fact, in most cases, lambda closures are not really necessary. (In TAO, a lambda expression creates an anonymous scope transparent function.)

Common:lambda is disposed of when a function definition or a top level form is preprocessed, as follows. (To make the interpreted code faster, TAO preprocesses functions when they are to be defined, thereby the runtime speed will be improved by about 20 to 50 percent.) In

```
(defun foo (x y) (mapcar #'(common:lambda (z) (list x z)) y)),
```

the lambda expression is "invisibly" expanded to

```
(closure '(x y) (lambda (z) (list x z)))
```

where the latter *lambda* is TAO native.

The "invisible" expansion is one of the most frequently used techniques for speed-up. Here, we sketch it briefly. A tag {evalcdr} is one of the invisible pointers in TAO. If the car of a list form is tagged with {evalcdr}, then the real car of the list form is the deferred car pointed to from the {evalcdr} and the cdr of {evalcdr} contains another form which should be evaluated instead of the original list form. In the usual list manipulation and I/O, the invisible expansion can not be seen by users, of course. (See Fig. 1.)

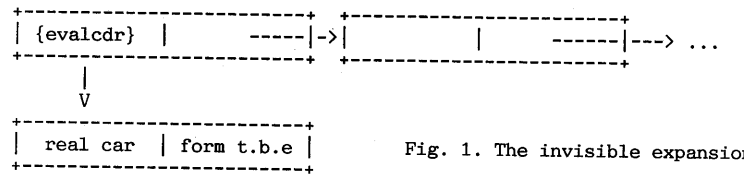

Fig. 1. The invisible expansion.

The macro expansion is a good example of the {evalcdr} expansion. The invisible expansion of *common:lambda* is of this kind, too. (There are other kinds of invisible expansions for making variable access quicker, storing information that is enough if once computed, and etc.)

If *common:lambda* has lexically detectable *go* expressions or *return-from* expressions inside which transfer the control to outside the closure, the expanded closure will be complicated a little. For example, in

```
(de foo (x)
    (prog (y p)
      (setq p (abracadabra x))
      (setq y (mapcar
                #'(common:lambda (z) (cond ((bar p z) (go a)) (t (qwe z))))
                x ))
      (return y)
      a (do-something-unusual) ))
```

the *common:lambda* expression is invisibly expanded, eventually, to

```
(closure '(x y p bas:lexical-$$-env)
         (lambda (z)
           (cond ((bar p z)
                  (sys:lexical-go (assq 'a (cdr bas:lexical-$$-env))
                                  '{the process identity} ))
                 (t (qwe z)) ))
         '(() (a)) )
```

where the last *(() (a))* indicates a pair of the lists of block names and go tags to be closed in the closure. (In this case, only one go tag "*a*" is present.) When the closure form is evaluated, these block names and go tags are sought at runtime, and recorded in the internal variable *bas:lexical-$$-env* with appropriate identifying numbers. And the corresponding

activation frame on the internal stack are marked with the corresponding identifying numbers in order to check the "extent". The function *sys:lexical-go* tests whether both identifying numbers of *sys:lexical-go* and its target coincide. It was quite lucky that there was room to store identifying number in the activation frame. At the very start of the design of TAO in 1981, we had decided the frame to include one word slot for debugging and miscellaneous use. If it were not, the implementation that does not load TAO any overhead would have been extremely difficult.

In the example above, TAO's native *lambda* will suffice, of course.

5. LOCAL FUNCTIONS

To make the TAO interpreter as fast as possible, some frequently used function symbols such as *car* and *cons* are allocated to the fixed addresses which correspond directly to the microcode entry addresses. This makes it impossible to associate a symbol with a function locally. However, TAO evaluates the car of a list form when it is not directly detected to be a function. Hence, TAO expands, say, an *flet* form invisibly as follows:

```
(flet ((foo (x) (bar x))) (foo y) (funcall #'foo foo))

=> (let ((foo-$$-localfn #'(common:lambda (x) (bar x))))
     (foo-$$-localfn y)
     (funcall foo-$$-localfn foo) )
```

Note that only *foo*'s which appear in the car of list forms or just after *#'* are expanded to *foo-$$-localfn*. (The compiler disposes of them differently, of course.)

6. CHARACTERS and STRINGS

TAO regards the string data type as, say, a "list" of characters, not as a vector of characters, though characters are packed byte by byte [3]. Hence, Japanese characters (16 bits) and ASCII characters (8 bits) can be freely mixed in one string. The TAO primitive operations on strings are *shead* (car of string), *stail* (cdr of string), *snull* (is it a null string?), *sequal* or *string-equal* (equality of strings), and *sconc* or *string-append* (append of strings). For this reason, null string *""* and a character *"a"* are also a string to the TAO user's view though they are represented by different internal data structures. (Microcode absorbs the difference.) This automatic coercion is quite analogous to the coercion between fixnum and bignum. (Why not character and string in Common Lisp?)

To accommodate TAO's string manipulation to Common Lisp's one, we added one bit, one-char-string-mode, to the system mode register by which TAO selects the action when a specified condition happens. If this mode is set on, *"a"* and *#\a* are discriminated strictly. However, it is still difficult to cope with the mixture of characters of different sizes. For example, how to do if one would want to alter an ASCII character in a string by a Japanese character? It is impossible to do so in most cases, since other string headers may point to inside the string. Hence, some destructive functions are left not to work on mixed character strings. This causes no problem in the TAO native string manipulation, since alteration of characters in a string always creates a new copy of string body. Anyway, there seem to remain problems in the specification of characters and strings in Common Lisp, especially for Japanese Lispers.

7. SEQUENCES and ARRAYS

It was not an easy problem how to merge TAO's original data types and Common Lisp's ones, especially for those related to sequences and arrays. It was relatively easy to make a code to detect something as a sequence by the multiway microbranch, though the code for sequences is the biggest among those newly written for Common Lisp. TAO's array is, however, a kind of function, not an inert data. For example,

```
(!tbl (array '(-10 10) 5))
```

sets a variable *tbl* to a two dimensional array whose first subscript ranges from -10 to 10 and second from 0 to 4. Array reference and array assignment is written as:

(tbl i j) and *(!(tbl i j) (foo x y z))* .

Hence, we had to add a code so that such expressions:

(aref tbl i j) and *(!(aref tbl i j) (foo x y z))*

also work correctly.

8. SETF

Most of the general assignment can be directly simulated by TAO's native assignment mechanism [4]. That is, most system defined general assignments can be simply replaced, or invisibly expanded to *(!position value)*. Only exceptions are *char-bit*, *ldb*, *mask-field*, **package** and **readtable**. The last two are specially dealt with, because each process which is an object in the Smalltalk sense holds them in its instance variables. The body of a readtable is attached to the process's stack to make the access as fast as possible, thereby the readtable reference for each character takes no more microstep than before where an embedded multiway microbranch directly dispatched characters. Hence, *setf* of **readtable** does some work other than a simple variable assignment, as well as **package**.

9. MULTIPROGRAMMING

Aluminum book does not address the problem of the multiprogramming and the selection between "deep binding" and "shallow binding" (page 38). TAO primarily adopts the multiprogramming and multiple-user environment [5]. It seems inevitable to choose the deep binding to cope with those. (By the way, how can we *makunbound* a special variable in the deep binding?) However, variable access in the deep binding is slower than the shallow binding. We added a new class of variables called "semi-globals" which will be accessed at a comparable speed to a vector element, and which will not occupy a portion of the symbol itself unlike "global" variable.

The "extent" problem of lexical go/exit closed in a lambda closure is not trivial if the closure is passed to other processes. We decided not to allow to go or exit outside the closure in any process other than the process which originally closed it. The second argument of *sys:lexical-go* described above is another key for the "extent" check.

There are also problems in the package systems if a package is shared by more than one processes or users. We slightly extended the package system to cope with the problem. It does not seem always possible that a package written in the strict Common Lisp specification can be shared.

10. OBJECT ORIENTED PROGRAMMING

The recent discussions on CommonLoops suggest that it will be embedded to Common Lisp in the near future. TAO, however, has adopted its own object oriented programming paradigm which is, in a sense, a chimera of Smalltalk, Flaver and LOOPS [6]. We had made some amount of programs and utilities which extensively utilize the TAO's object oriented programming. Fortunately, since message passing form in TAO is represented by a bracketed expression of the form:

[receiver message argument ...]

which does not contradict to Common Lisp. It can be said there is still room to accommodate CommonLoops in TAO, if needed. That is the TAO's spirit!

11. PERFORMANCE

As can be easily suspected, Common Lisp on (or in) TAO is not slow compared with the native TAO. Only one major factor that may slow down the runtime speed in the interpreter is *common:lambda*. A simple experiment shows that the closure creation is rather slow but the execution is a bit faster than *bas:lambda* which has to seek free variables dynamically.

Totally, Common Lisp can run as fast as TAO both in the interpreter and in the compiled code. That is, 20 to 50 times faster than VAX Lisp on VAX/8600 in the interpreter, and 2 to 4 times faster in the compiled code [7].

12. CONCLUSION

Implementing Common Lisp on TAO (or, effectively in TAO), we think, proves that TAO has a potential for ever evolution to encompass everything useful and fascinating. We are satisfied with the achieved performance and degree of compatibility. It can be said that the Lisp part of TAO is almost settled now. We continue on further improving TAO in the directions to object oriented programming, logic programming and their combination in order to reach the goal of really harmonious multiple programming paradigm.

ACKNOWLEDGMENT

This work should not by any means be credited only to the author in the title. It is the result of a considerable amount of effort of the following persons: Hiroshi Gomi, Shuji Jimbo, Minoru Kamio, Katsumi Kishida, Hiroshi G. Okuno, Mitsuru Ooi, Kunio Oono, Nobuyasu Osato, Yoshihiro Shintani, Hiroyuki Sugiyama, Kyoji Umemura, and Kazunori Yamamori. We all thank many other persons, especially Yasushi Hibino, Takashi Sakai and Yasuhiro Yamada, who helped checking the code and specifications, set the working environment and gave us encouragement.

REFERENCES

[1] G.L. Steel Jr.: COMMON LISP The language. Digital Press, 1985.
[2] H.G. Okuno, I. Takeuchi, N. Osato, Y. Hibino and K. Watanabe: "TAO: A Fast Interpreter-Centered System on Lisp Machine ELIS" ACM Conference on Lisp and Functional Programming, 1984.
[3] T. Sugimura, H.G. Okuno and I. Takeuchi: "Japanese String Manipulation in TAO". IPSJ WGSYM 36-4, 1986 (in Japanese).
[4] N. Osato, I. Takeuchi and H.G. Okuno: "A General Assignment Mechanism in TAO". IPSJ WGSYM 31-2, 1985 (in Japanese).
[5] I. Takeuchi, H.G. Okuno and N. Osato: "On the operating system on TAO". IPSJ WGOS 24-8, 1984.
[6] N. Osato, H.G. Okuno and I. Takeuchi: "Object-Oriented Programming in Lisp". IPSJ WGSYM 26-4, 1983.
[7] H.G. Okuno: "The Report of The Third Lisp Contest and The First Prolog Contest". IPSJ WGSYM 23-4, 1985.