記 号 処 理 48－2
（1988. 11. 29）

# UtiLispの新しい実現手法

金子 敬一
東京大学 工学部

UtiLispはメインフレームを始めとし、ミニコンピュータからパーソナルコンピュータまで幅広く稼働するlispの処理系である。ただし、通常のパーソナルコンピュータではMotorola MC68000ボードを必要とするのが問題点であった。今回はMC68000用UtiLispのコードを展開し、多くのパーソナルコンピュータがCPUとして採用しているINTEL i8086の命令で置き換えてUtiLispをパーソナルコンピュータ上で稼働させる手法を提案する。また、実際に幾つかの計算機上で検証した結果から、その実現、移植性、性能についても報告する。

# A New Implementation Method for the UtiLisp System

Keiichi Kaneko

Faculty of Engineering, University of Tokyo
7–3–1, Hongo, Bunkyo-ku, Tokyo, 113 Japan

UtiLisp is one of lisp dialects which is available on mainframes, minicomputers, and personal computers. However, as for personal computers, UtiLisp needs an extra Motorola MC68000 board, which is sometimes expensive only for this purpose. Therefore a new implementation was designed for the INTEL i8086 series which are used widely for personal computers. This paper proposes this new method and reports the results of actual implementations on several machines.

## 1. Introduction

UtiLisp (University of Tokyo Interactive LISt Processor) was originally implemented by Chikayama [1] for mainframe machines. Then Tomioka *et al.* rewrote the code for MC68000 [5, 6, 7]. Let us refer to this version of UtiLisp as 'UtiLisp68.'

Since these implementations depend on the machine architectures deeply, a new UtiLisp was designed to fit in with all 32-bit machines [3]. It is called 'UtiLisp32.'

According to the flow shown in figure 1–1, UtiLisp now works on mainframes, minicomputers, and even personal computers. However, as for personal computers, UtiLisp requires an extra MC68000 board which is sometimes expensive only for this purpose. Still another implementation was, therefore, newly designed for the INTEL i8086 series which are used widely for personal computers.

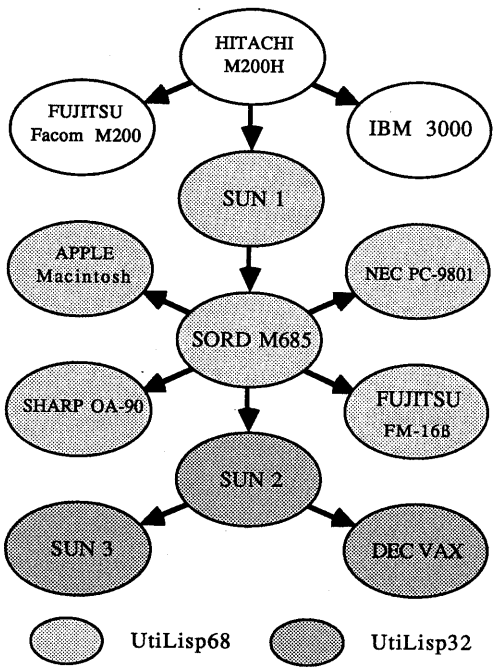



Figure 1-1 UtiLisp Family

This paper proposes this new method and also reports the results of actual implementations on a couple of machines. UtiLisp for the i8086 series is lacking several features which other UtiLisp systems have. For example, it does not support two lisp objects ('flonum' and 'bignum'). So, let us call it 'cUtiLisp' which is an abbreviation for 'compact UtiLisp' to make distinction from others.

## 2. Basic Design

cUtiLisp basically emulates a system designed for 32-bit computers by translating the code to fit in with the i8086. First UtiLisp32 was selected as a candidate system for its flexibility. And the following changes turned out to be necessary:

- Taking account of the segmentation scheme of the CPU, it is natural to allocate same type of lisp objects in a same segment. But it causes one restriction that we can not allocate an area which is more than 64 kilobytes for each object except for 'cons' which is treated in a special manner. The way of allocation, on the other hand, makes the type checking easy. That is, because same type of lisp objects are accessed by their unique segment and various offsets, the segment information represents the object type and works as the pointer tag.

- The initial target machine was NEC PC-9801 with 640 kilobytes main memory. Its operating system, MS-DOS, and two interface boards occupy considerable amount of the memory. Often occurs the same situation on the other personal computers. So it was decided to eliminate lisp objects 'flonum' and 'bignum' to save the code area to process these ob-

jects. Now it is under way to transport the cUtiLisp system to Fujitsu FM-R70 whose operating system is XENIX. On this machine, cUtiLisp works under the i80286 protected mode which allows huge address space. So, the eliminated objects are going to be added back to the system.

- The original code for UtiLisp32 bases on the existence of fifteen or sixteen 32-bit registers, while the i8086 has only eight 16-bit registers. So 32-bit registers, except for stack pointer and frame pointer, are allocated on memory and used as pseudo registers. Accesses for these registers are so frequent that the data segment register 'ds' is selected to hold the value of this segment.

However, the first change is not compatible with UtiLisp32 which utilizes the object tag scheme in several parts, while UtiLisp68 adopts the pointer tag scheme completely. As a result, cUtiLisp became a mixture of UtiLisp68 and UtiLisp32. That is, cUtiLisp emulates a new system which is a combination of those two systems.

The code translation proceeded followingly. Original codes for UtiLisp68 and UtiLisp32 are both written in LAP (Lisp Assembly Program) form. For example:

```
evandret
        (pea    return)
eval
        (iflist  A evrec)
        (ifnotsy A evret)
        (valuea)
evret
        (rts)
```

is the entrance part of evaluator. So, first of all, it was necessary to rewrite LAP expander to generate the i8086 assembly code which is accepted by Microsoft MASM assembler. The above code was expanded as:

```
evandret label near
         push    cs
         mov     ax, offset return
         push    ax
eval     label near
         mov     ax, word ptr [A+2]
         iflist  ax, evrec
         mov     ax, word ptr [A+2] — (†)
         ifnotsy ax, evret
         les     bx, dword ptr [A]
         mov     ax, word ptr es:[bx+2]
         tst     ax
         jnz     L0000
         jmp     near ptr ubverr
L0000    label near
         mov     word ptr [A+2], ax
         mov     ax, word ptr es:[bx]
         mov     word ptr [A], ax
evret    label near
         retf
```

where instructions such as 'iflist', 'ifnotsy', 'tst' and 'retf' are defined as macros elsewhere. And after this stage, the redundant instruction which is marked with dagger (†) was omitted by hand optimization. This optimization is mainly based on usual techniques which, for example, include the eliminations for common subexpressions, chain branches, and loop invariants and the peephole optimization. The translation basically proceeds like this. However the critical changes as mentioned above were necessary. The sections below explain them in detail.

## 3. Evaluator

### 3.1 Memory Configuration

Figure 3–1 shows the memory map of cUtiLisp. Note that there are a few constraints caused by the lambda binding mechanism. 'Bound symbol' objects appear in the stack area and are represented by setting the highest bit of the symbol segment according to the manner of the UtiLisp32 system. To distinguish it from 'fixnum' whose highest two bits are set, the symbol segment must be less than 0x4000 in unsigned comparison. And the stack segment must be less than 0x8000 in unsigned comparison not to confuse a bound symbol with some stack frame. Satisfying these two restrictions is the condition to support the cUtiLisp system.

### 3.2 Type Checking

Figure 3–2 shows all lisp objects of cUtiLisp. And figure 3–3 shows the heap area of memory map in detail.

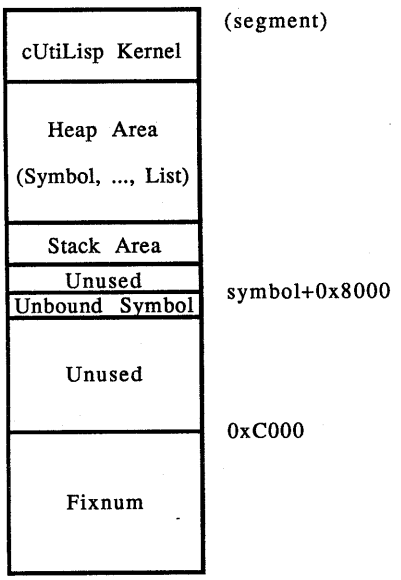



Figure 3-1 cUtiLisp Memory Map

Each lisp pointer consists of a segment and an offset. Then it is easy to check the type of objects: 'symbol', 'string', 'code piece', and 'stream' in a segment comparison. A reference is a lisp pointer which directly points to one element of a vector (see figure 3–4) and accordingly has the 'vector' segment. So, it is a problem to know whether a lisp pointer which has the 'vector' segment is a 'vector' or a 'reference'. The answer is: if the segment

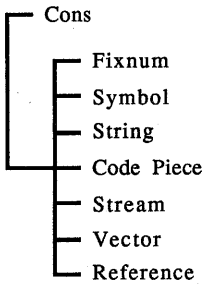



Figure 3-2 cUtiLisp Objects

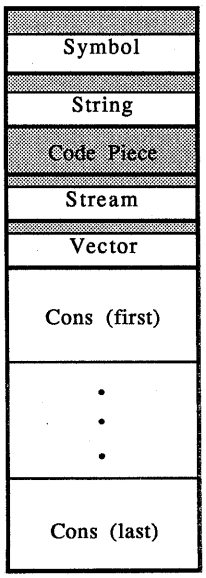
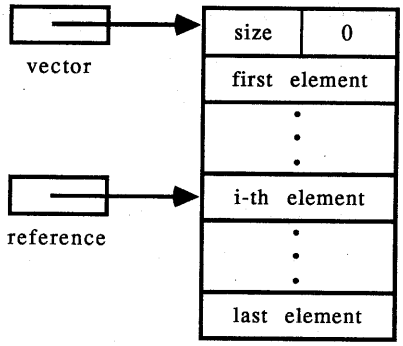



Figure 3-3 cUtiLisp Heap Area




Figrure 3-4 Vector and Reference

part of the cell which the pointer is pointing to is zero, then the pointer is a 'vector', otherwise it is a 'reference'. 'Fixnum' is a special object which is coded in a pointer. It is a signed 28-bit integer. The highest two bits of its segment part are set on for the type checking and the lowest two of its offset part are reserved for the garbage collection. That is, a lisp pointer is a 'fixnum' if and only if the segment is greater than or equal to 0xC000 in unsigned comparison. Users can augment the area for 'cons' in blocks of 64 kilobytes as much as possible. Though 'cons' objects occupy more than one segment, these segments are continuous and only one signed comparison can check the type for 'cons' (or 'atom'). That is, if the segment of the pointer is less than the list segment, it is an 'atom', otherwise it is a 'cons'.

### 3.3 Lambda Binding

cUtiLisp uses the stack area for the lambda binding mechanism. Figure 3–5 illustrates this mechanism. For every function call, it is necessary to store the values of lambda symbols and set new values to them on entrance, and to restore the old values on exit. So, for each symbol, its value is pushed on the stack. Then the pointer to the symbol is also pushed on the stack with setting on the highest bit of the segment part. To undo the lambda bindings, cUtiLisp searches the stack area to find a cell whose highest two-bit pattern is '10'. It is peculiar to bound symbols. Then cUtiLisp pops the pointer to the lambda symbol with setting off the highest bit of the segment part, and restores the old value to the symbol by poping it from the stack area.

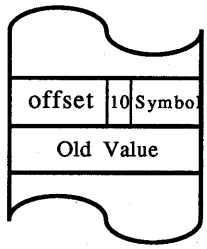



Figure 3-5 Lambda Bindings

### 4. Garbage Collector

cUtiLisp adopts a compaction-type garbage collection as well as UtiLisp68 and UtiLisp32 do. The garbage collector consists of marking phase and compaction one. The former marks the alive objects recursively using the preordered marking technique. As this marking phase consumes the stack area for recursion, it is to be shifted to adopt a reversed link method.

The compaction phase uses Morris' algorithm [4]. The algorithm, generally, is not good at coping with a several objects such as 'string' and 'code piece' which are variable-length and have non lisp pointer cells. The reason is as follows: The garbage collector traverses the heap area from both ends. The moment it visits a cell, it is necessary to know whether the cell contains a lisp pointer or not. When it approaches a variable-length object from the side which does not have a tag, it is impossible to know the contents of the object.

The garbage collectors of UtiLisp68 and UtiLisp32 exchange the object tag and the content of the last cell for each variable-length object in preparation for the reversed order traversal. Fortunately, objects of cUtiLisp are allocated separately according to their types. Each 'string' object has no lisp pointer, and each 'code piece' object has a lisp pointer which has another segment value. It suffices, therefore, to traverse the heap area twice from same direction and this brings the same effect as if one of the traversals were from opposite one.

#### 5. Portability

Though cUtiLisp was developed on the NEC PC-9801 machine, it is machine independent. Therefore it works on every computer whose CPU belongs to the i8086 series and whose operating system is MS-DOS, provided that the following two conditions are satisfied:

- The symbol segment is less than 0x4000 in unsigned comparison.
- The stack segment is less than 0x8000 in unsigned comparison.

These restrictions are somewhat severe for a personal computer with RAM disk and/or interface boards. Section 7 describes the relaxation of the constraints. Currently, cUtiLisp works on NEC PC-9801, EPSON PC-286, and IBM-5550 machines in our laboratory.

It is under way to transport the cUtiLisp system to a XENIX machine which supports Microsoft MASM assembler. The target machine is Fujitsu FM-R70. In this environment, cUtiLisp works under the i80286 protected mode. Each segment, therefore, has no relation with the actual memory of the machine. It is a very little number called a segment selector. So the previous two restrictions was redundant in practice. As a result, it sufficed for transportation to rewrite the interface parts between cUtiLisp and the operating system.

#### 6. Performance

Timings were measured for a typical benchmark test tarai-5 on several machines. The definition of tarai is as follows:

```
(defun tarai (x y z)
  (cond ((> x y) (tarai (tarai (1- x) y z)
                        (tarai (1- y) z x)
                        (tarai (1- z) x y)))
        (t y)))
```

Table 6–1 denotes the number of function calls in tarai-5 for each function. Table 6–2 shows results of execution. The execution performance of cUtiLisp is tolerable in comparison with other lisp systems designed for personal computers considering that cUtiLisp emulates the UtiLisp system for 32-bit machines.

Table 6-1 Number of Calls in Tarai-5

| function | times |
|---|---|
| tarai | 343073 |
| cond | 343073 |
| > | 343073 |
| 1- | 257304 |

Table 6-2 Timings of Tarai-5

(times are all in second)

| Machine | CPU/Clock | tarai-5 | OS |
|---|---|---|---|
| EPSON PC-286 | V30/10 | 240 | MS-DOS |
| Fujitsu FM-R70 | i80386/16 | 70 | XENIX |
| NEC PC-9801VX | i80286/10 | 153 | MS-DOS |
| IBM 5550 | i8086/8 | 349 | MS-DOS |

## 7. Results and Further Improvement

By rewriting the LAP expander and the simple hand optimization, it was easy to implement a subset of UtiLisp, cUtiLisp, for the i8086 series. Currently, the following points are considered for improvement:

- Achieving high portability
- Gaining execution performance
- Upgrading cUtiLisp to a full set version of UtiLisp.

To achieve high portability, two changes are possible. First comes the change of the lambda binding mechanism. The current cUtiLisp system represents a bound symbol by setting the highest bit of symbol segment according to the manner of UtiLisp32. If a new segment is introduced for bound symbol segment, there is no need for symbol segment to be less than 0x4000 in unsigned comparison. As for stack segment, a stack frame is never confused with a bound symbol. Therefore, the previous two restrictions becomes only one:

- The stack segment is less than 0xC000 in unsigned comparison.

This restriction is necessary because a stack frame would be confused with a fixnum without it. Second comes the change of fixnum representation by introducing a new segment for fixnum for type checking. This change makes the above constraint unnecessary. In this case, however, a fixnum becomes a signed 14-bit integer. So bignum object must be introduced for compensation.

To gain performance, since one of the pseudo registers is used much more frequently than others, it is able to assign a two 16-bit actual registers for the pseudo one. This will reduce the times of memory accesses. And thorough code optimization will result in a better performance.

Roughly, cUtiLisp lacks the following features:

- Object types for bignum and flonum
- Compiler
- Attention handler
- Linkage ability for external programs.

Since cUtiLisp is in evolutionary cycles, it is postponed to implement the latter three features. The first one has a large relation with the portability and its implementation is also prolonged for the time being.


### Acknowledgement

I wish to thank Professor Eiiti Wada and Associate Professor Masato Takeichi for giving me a chance to transport cUtiLisp to FM-R70. And I also wish to thank Research Assistant Yutaka Tomioka and Mr. Akitoshi Fujiwara for their insightful comments on an earlier draft of this paper.

**References**

[1] Chikayama, Takashi: "Implementation of the UtiLisp System," *Trans. IPS Japan*, IPS Japan, Vol. 24, No. 5, pp. 599–604, 1983 (In Japanese).

[2] Chikayama, Takashi: *UtiLisp Manual*, Technical Reports METR 81-6, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, Univ. of Tokyo, Sept. 1981.

[3] Kaneko, Keiichi and Kei Yuasa: "A New Implementation Technique for the UtiLisp System," *Preprints of WGSYM Meeting*, IPS Japan, Vol. 41, Jun. 1987.

[4] Morris, F. Lockwood: "A Time- and Space- Efficient Garbage Compaction Algorithm," *CACM*, Vol. 21, No. 8, pp. 662–665, Aug. 1978.

[5] Terada, Minoru and Eiiti Wada: "Transportation of Object Files between the Systems with Common CPU," *Preprints of WGSW Meeting*, IPS Japan,Vol. 87, No. 11, pp. 89–95, Feb. 1987 (In Japanese).

[6] Yuasa, Kei and Keiichi Kaneko: "Transportation of UtiLisp to Macintosh, the Days of Hardship," *Proceedings of the 27th Programming Symposium*, IPS Japan, pp. 131–141, Jan. 1986 (In Japanese).

[7] Wada, Eiiti and Yutaka Tomioka: "Transportation of UtiLisp to 68000," *Preprints of WGSYM Meeting*, IPS Japan, Oct. 1984 (In Japanese).