記 号 処 理 41 − 7
(1 9 8 7. 6. 2 6)

UtiLispの新しい実現法

金子敬一 ・ 湯浅 敬

東京大学工学部

UtiLispはメインフレーム上のLisp処理系であり、コードの書換えによってMC68000をCPUとする計算機上で稼働している。これらの計算機は32ビットのレジスタを持ち、アドレスバスは24ビットで上位8ビットは使用しない。今までのUtiLispはこの点を利用し、ポインタタグ方式により実行速度を上げている。しかしながらこの方式のために32ビットのアドレスバスを持つCPU上にUtiLispを実現することが困難であった。我々はこの問題点を解決し、より広範囲の計算機に対応できるUtiLispの実現法を提案し、実際に、MC68010,MC68020,Vaxといった32ビットアドレスのCPU上で検証した結果から、その実現、移植性、性能について述べる。

A NEW IMPLEMENTATION TECHNIQUE FOR THE UTILISP SYSTEM

Keiichi Kaneko and Kei Yuasa

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 and MC68000 CPU machines. We implemented new UtiLisp for 32 bit address CPU (MC68010, MC68020, Vax). There is a determinate difference between architectures of them and those of old CPU's. So we had to make a new design of this language.

This paper denotes this difference of architectures and then describes about the implementation, the transportability and the performances of this new UtiLisp.

1. Introduction

1-1 Motivation

UtiLisp was originally implemented for Hitachi M200H in CCUT (Computer Centre, the University of Tokyo) by Chikayama[1]. Tomioka rewrote the code for Motorola MC68000 whose architecture resembles that of IBM360. We call this version UtiLisp68[7]. And we have transported these UtiLisp systems on several computers as in fig. 1.1[6].

These systems take advantage of the architecture of the CPU's so that they could achieve high execution performance. IBM360 and MC68000 have 32-bit registers and 24-bit address bus (the highest 8 bits are not used). This enables pointer tag scheme. But this is also a reason why machines on which UtiLisp is availabe are restricted.

Today, most Unix machines and workstations have 32-bit address bus. On these machines, we can not use the pointer tag scheme because the highest 8 bits are also parts of the address. Thus we decided to make a new design of UtiLisp which is available on 32-bit CPU. We call this one UtiLisp32.

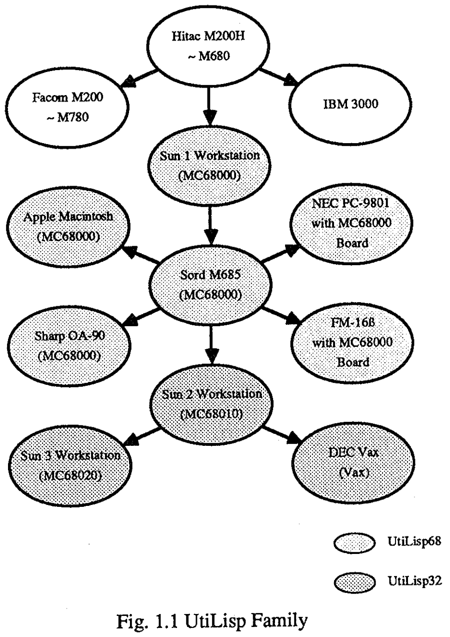
Hitac M200H ~ M680
Facom M200 ~ M780
IBM 3000
Sun 1 Workstation (MC68000)
Apple Macintosh (MC68000)
NEC PC-9801 with MC68000 Board
Sord M685 (MC68000)
Sharp OA-90 (MC68000)
FM-16β with MC68000 Board
Sun 2 Workstation (MC68010)
Sun 3 Workstation (MC68020)
DEC Vax (Vax)
UtiLisp68
UtiLisp32

Fig. 1.1 UtiLisp Family

1-2 Basic Design

There are some feasible methods to make a lisp system for machines on which we can not use pointer tag scheme.

(1) Masking tag on addressing
(2) Object tag (tag on objects)
(3) Division of heap area according to type

Each one has merits and demerits. The former two can use heap area effectively at the sacrifice of execution speed. They need too much time on addressing or type checkings. (3) can make a better performance than them, but it might cause inefficiencies in memory management as follows:

- Division with improper proportion invokes the garbage collector frequently.
- The area allocated for a certain kind of unfixed-size objects such as vectors and strings, restricts the maximum size of them.

It is difficult to estimate the proportion of the size of the area for each lisp object in advance. And it is more difficult to change the border dynamically according to the circumstances.

The way we selected for UtiLisp32 is the combination of (2) and (3). We first picked up the fixnums, the symbols and the lists out of all UtiLisp objects and attached importance to them. This is because we know from our experiences that the performance of Lisp evaluator depends on type checkings especially for the lists (**consp**, **atom** etc) and for the symbols (**symbolp**). As for other objects, we allocated them with their object tags in the same area.

In the following chapters, we denote the interpreter, the compiler and the garbage collector of UtiLisp32 in this order in comparison with those of UtiLisp68, then discuss the portability and the performance of them.

2. Interpreter

2-1 Memory Allocation

UtiLisp supports the lisp objects as in fig. 2.1.1. The bignums are specific to UtiLisp32. Figure 2.1.2a shows the initial state. UtiLisp32 has all the predefined objects in its data area, then it allocates a fixed heap area and a heap area.

Then UtiLisp32 divides the heap area into three parts. They are for the symbols, the lists and the other objects respectively. The predefined objects will be located on setting up into the proper areas (see fig. 2.1.2b). The fixed heap area is to contain compiled codes. We will mention the compiler in the next chapter.

2-2 Type Checks

UtiLisp does not make a box for small integer in the heap area but codes it in a pointer. So UtiLisp32 must be able to distinguish the fixnums from the other objects allocated in the heap. UtiLisp32 uses the highest two bits of a pointer for this distinction.

If the highest 2 bits are '11', it is not a pointer but fixnum. If they are '10', it is an object tag for 'others'. Otherwise, if they are '00' or '01', it is a pointer into the heap area. Here we assume that the heap will be allocated where the address is less than 0x80000000.

UtiLisp32 also reserves the lowest two bits for marking of garbage collector. As UtiLisp allocates the lisp objects on four bytes boundary, the lowest two bits of the lisp pointers are always zeros. So, as for a fixnum, the rest 28-bit field contains the number itself.

We allocated 'nil' at the bottom of the symbol area, adjacent to the others area. And one of registers holds the address of it. So, to check if a pointer is a symbol or not, it suffices to make an unsigned comparison of the pointer with the register which contains the value 'nil'. Notice that the fixnums, whose MSB's are always on, are greater than 'nil' in unsigned comparisons.

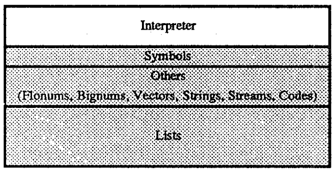

Fig. 2.1.2a UtiLisp32 Initial Memory Map

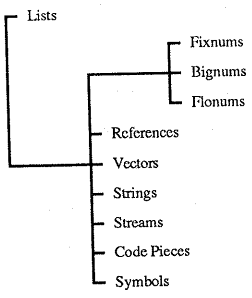

Fig. 2.1.1 Lisp Objects

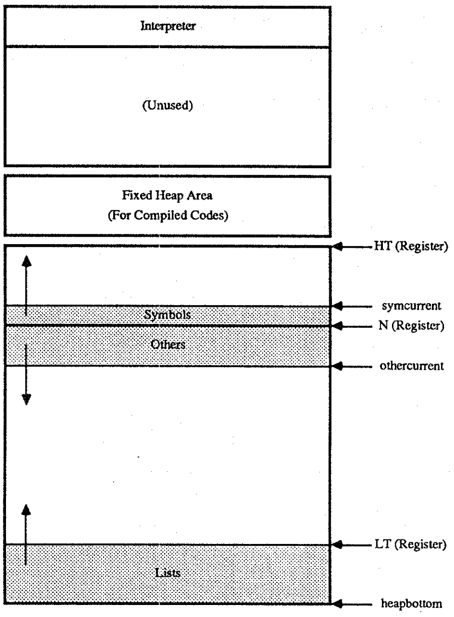

Fig. 2.1.2b UtiLisp32 Run Time Memory Map

Another register points the top of the list area to perform the type checks of the lists efficiently. In this case, UtiLisp32 makes signed comparisons. This time, the fixnums are less than the register because they are always negative.

The objects in the 'others' area have 32-bit object tags. The highest bits of the tags are always '10'. UtiLisp32 takes advantage of this fact for the type checkings. There is no object whose highest two bits are '10'. So, if a pointer points the object whose first cell shows this pattern, it must be 'others' object. If a lisp pointer points in the others area without appropriate object tag, then UtiLisp32 regards the pointer as a reference to a vector element.

2-3 Argument Passing

On calling functions UtiLisp32 uses a stack to pass the arguments to subroutines after the evaluation of them. The layouts of stack frames are almost the same as UtiLisp68. Each stack frame includes local variables, return addresses, a code base, lambda bindings and a dynamic link to the previous frame. UtiLisp32 passes the last argument in a register to lessen the loss of memory accesses. If the routine takes no argument, the value of the register is undefined.

2-4 Lambda Binding

UtiLisp68 has implemented the shallow binding by storing a pair of the bound symbol and the old value on stack using the binding tag, '0xB0' (see fig. 2.4.1a). On returning from the function call, UtiLisp68 sweeps the stack area and finds the binding tag. Then it pops up the lambda variable and restores the old value in the value cell of it. UtiLisp32 sets the highest bit and resets the second highest to indicate the binding structures (see fig. 2.4.1b). We can identify these tags in comparison with value '0xC0'. In this case, the highest bit clearance is also necessary for the recovery of the old value in addition to the UtiLisp68 process.

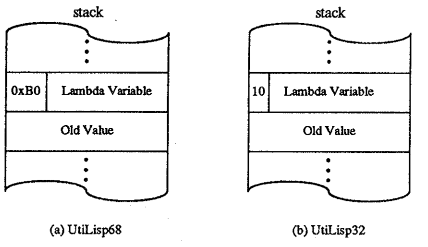

Fig. 2.4.1 Lambda Bindings

3. Compiler and Garbage Collector

3-1 Compiler

Differences of the compiler for the MC680x0 version of UtiLisp32 from that of UtiLisp68 are mainly due to those of the interpreters. Major changes are as follows:

- Because UtiLisp32 passes the last argument in a register, we changed the interface between the compiler and the interpreter.
- In case of UtiLisp68, only one comparison of the pointer tag is sufficient to perform the type check. But in case of UtiLisp32, it was necessary to generate codes for the type check in accordance with the object type.
- It was also necessary to change the routine which expands built-in functions for the fixnums, such as '*', '1-' and '>=', into inline codes, because of the difference of the format of the fixnums in UtiLisp32.

Except for these changes, almost all the routines of the compiler for UtiLisp68 were available for the MC680x0 verion of UtiLisp32 compiler.

We have also implemented the Vax-version compiler. This time, it was necessary to make a code generator (assembler) for Vax. As the assembler we made is simple and straight, there is much room for improvement.

3-2 Garbage Collector

We adopted the compaction-type garbage collection for UtiLisp32 as well as UtiLisp68. UtiLisp32, however, uses the heap division and the object tag scheme against UtiLisp68 which uses only the pointer tags for the type checks. Thus we added several changes to the fundamental algorithm.

The garbage collector uses the lowest two bits of the pointer for mark and stop bits. The marking phase uses the preorder traversal method and it also uses the reversed link technique to avoid using the stack. The most difficulties in this phase is a process for the reference objects (direct pointer to vector elements). It is necessary to rewrite the object tag of the vector so that we can know which element the reference points. Then garbage collector sets stop bit in the tag area to indicate that the present vector is marked by the reference.

The compaction phase follows the marking one. UtiLisp68 uses Morris' algorithm[4]. UtiLisp32 uses the same one for the compaction of the symbol and the list areas. However, the direction for compaction of the others area differs from those of these two areas. So, we applied Knuth's algorithm[3] for this area. That is, , UtiLisp32's garbage collector uses the two algorithms properly according to the area to compact.

4. Portability

UtiLisp32 and UtiLisp68 are both written in LAP (Lisp Assembly Program) form[5][7]. Macro ability of LAP codes achieves high readability and efficient source code control. When we make these UtiLisp systems, first we use some lisp system and expand these LAP codes in accordance with assembler syntax of target machines. Then we assemble the expanded codes using assemblers on the machines.

In case of transportation, we simulate this process. There are two ways which fig. 4.1a and fig. 4.1b illustrate. We usually adopt the former one, that is, the way which expands the LAP codes on the host machine using UtiLisp system, and downloads them onto the target machine, then assemble them using the assembler on the target. The latter is available when download process takes much time. It performs download of the LAP codes onto the target machine, and proceeds the transportation using other lisp system there. Anyway, when UtiLisp system itself on the target machine can expand the LAP codes, the transportation moves a new stage which fig. 4.1c shows. This time, in case which fig. 4.1b shows, it is necessary to make a LAP expander for UtiLisp system.

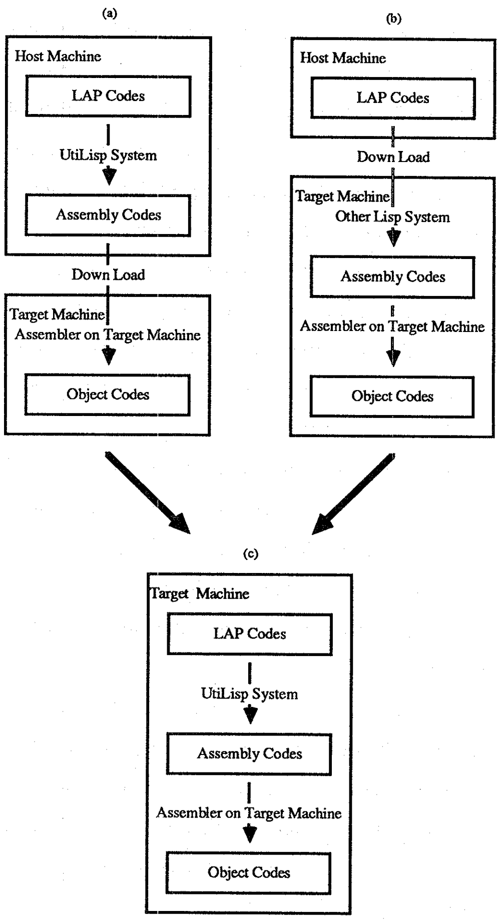

Fig. 4.1 Transportation Process of UtiLisp Systems

We have implemented UtiLisp32 for Vax, whose architecture resembles MC680x0, rewriting the source codes for Sun Workstations. UtiLisp32 consists of about 15000 lines of LAP codes. Except for the changes of about 300 lines of the machine dependent parts, following processes were necessary:

- We changed the LAP expander for MC68000 instructions to generate according Vax ones,
- And rewrote the memory accessing parts taking account of the differences of endian-ness between MC68000 and Vax.
- We processed instructions, which MC680x0 supports but Vax does not, using macro ability to substitute them by several instructions or changing the source codes directly.

As a result, we finished implementation in fifteen man-days.

Originally, LAP was made in order to expand a large quantity of macros for assembler without macro expander. But this is also useful for transportation of the code between common CPU[5][6]. And this time, we succeed in using LAP for transportation between different CPU's, MC680x0 and Vaxen.

5. Performance

5-1 Preliminaries

We measured timings of a typical benchmark test on UtiLisp and other lisp systems.

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))).
```

And we measured timings of (tarai 10 5 0). This test calls 'tarai' 343073 times, 'cond' 343073 times, '>' 343073 times, and '1-' 257304 times. Table 5.1.1 shows the results of this benchmark test. Hyphens in the table indicate omissions of measurement. Numbers followed by daggers indicate the results without type checkings.

Table 5.1.1 Timings of Tarai-5 on Lisp Implementations

(times are all in seconds)

System	Machine	(tarai 10 5 0)	
		Interpreter	Compiler
UtiLisp	Hitac M682H	3.82	0.233
UtiLisp32	Vax-8600	23.8	2.79
	Sun 3 / 260	22.0	2.10
	Sun 3 / 52	53.0	4.70
	Sun 2 / 120	117	13.5
	Sord M685	108	——
UtiLisp68	Sord M685	104	11.4
	NEC PC-9801	145	16.2
	Apple Macintosh	214	——
Franz Lisp	Vax-8600	81.3	31.6†
	Sun 3 / 52	262	86.1†
	Sun 2 / 160	499	159†
Symbolics 3640	Symbolics 3640	369	3.85
KCL	Vax-8600	393	8.38†

† : No Type Check

5-2 Comparisons with UtiLisp68

Sord M685 supports both UtiLisp68 and UtiLisp32 systems. Comparison of two on the machine shows that UtiLisp32 is inferior to UtiLisp68 by several percents. Among the differences of UtiLisp32 from UtiLisp68, the type checkings for the fixnums and the symbols work advantageously to UtiLisp32. Improvements on the argument passing method also gains the execution performance. However, the others, that is, the type checkings for the other objects, the processes for the lambda bindings, and the operations of the fixnums, cause downfall of performance. We think this is the reason of the inferiority of UtiLisp32.

5-3 Comparisons with Other Lisp Systems

The design of UtiLisp intends high performances of both interpreter and compiler. Some lisp systems especially ones that use lexical binding scheme make much of only the compilers. In these systems, the interpreters are only debuggers. So, it might be meaningless to compare the performance of the interpreters.

Compared in compiler, Utilisp32 is ahead of Franz and KCL. One reason is that UtiLisp is fully written in assembler language while others in C. Of cource, each compiler generates good assembly codes, but compiled codes sometimes call routine in interpreter (kernel).

Besides, UtiLisp allocates the Lisp objects so that the type checking would be done as soon as possible. This design enabled the high performance of Utilisp32.

6. Conclusions

In this implementation, we designed a new UtiLisp system which fits in with much more computers with a little loss of execution performance. In addition, we achieved high portability using the LAP.

Acknowledgments

We wish to thank Professor Eiiti Wada for his guidance as the supervisor, and Research Assistant Minoru Terada for his helpful advices. We also wish to thank Dr. Michio Kimura and Mr. Hideya Iwasaki for their insightful comments on an earlier draft of this paper, and Mr. Toshinari Takahashi, Mr. Makoto Ishii and Mr. Hitoshi Tominaga (Fujitsu), for making a good job on implementation of UtiLisp32.

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 Instrumental Physics, Faculty of Engineering, Univ. of Tokyo, Sept. 1981.

[3] Knuth, Donald E.: *The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1969.

[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, pp. 15-21, Oct. 1984 (In Japanese).