Invited Paper

History of UtiLisp Hacking

EIITI WADA*

The history of UtiLisp hacking is presented. UtiLisp is a dialect of MacLisp. In these ten years, a number of UtiLisp interpreters and compilers have been developed by the students of our group. Each time, new methods were employed and practical systems were put into operation. UtiLisp 360, the ªrst ancestor of the series and the system for main frame machine was coded in macro assembly language; UtiLisp 68 for workstation was translated by macro expansion deªned in UtiLisp. UtiLisp 32 had both address and object tags; UtiLisp/C checks data types by means of low order bit tags. The experience of implementations with given language speciªcation except the ªrst one, exposed students in the environment where they considered the implementation method for relatively large system in front of the new machines. This was a typical on the job training.

1. Introduction

It is often the case that around quality software system programmers gather to improve the system ever higher than before. What follows is one of such examples; in this case, the group of programmers are the graduate students of a software laboratory.

In these ten years, over 30 students joined to the author's laboratory, worked for two to five years and left. It was just like the comets that visit our solar system, stay a while and then leave. While the students were with us, most of them interacted with a UtiLisp system, a dialect of Lisp, some very intensively (installed the system; transported them to other computer; made improvements, etc.), others rather weakly (just as occasional users and critics) just as comets that pass very closely around the sun, while others pass quite far from it.

In these ten years, the central activity of our group happened to be around UtiLisp although it was not foreseen. Many students expressed their interest in developing our own Lisp system of high quality in fairly stable programming style, based on the philosophy of our laboratory; thus strengthening the membership of the group. In developing four generations of UtiLisp systems, students learnt key issues in system program design, learnt problem solving attitudes, programmed difficult parts with concentrated mind and stamina, improved the code to increase efficiency and maintenability, and acquired manual or document writing techniques. They experienced four life cycles of UtiLisp systems.

When the author was still a student around 1960, at Professor Hidetosi Takahasi's laboratory where we built the first and second Parametron Computers PC1 and PC2, all the graduate students enjoyed watching life cycles of computer systems in a university laboratory. We observed all phases of the computer system development from the circuit element design, logical circuit design, architecture design, memory design, assembly language design, writing many basic programs, including initial orders, debugging programs and programs for hardware maintenance. Presently in a software laboratory like ours, almost infinitely more reliable hardwares are supplied by computer manufacturers with convenient system development tools. With the development of computer network systems, advanced tools are arriving from other similar laboratories across the net. Accordingly, software laboratories need identity around which they may show their own programming style and philosophy.

Examples of such identities include Unix, smalltalk, GNU etc. In our case, it was UtiLisp although we admit that ours is not so idiosyncratic as other examples quoted above. Nevertheless we found that it played an essential role in software engineering education in displaying students development of a typical software product. New students begin to learn how to read source program lists of UtiLisp system while they are studying other computer science materials. Gradually they proceed to the stage where they rewrite small part of the system and learn how to use the operating system, how to tackle huge system programs. This seems to be one of the ideal courses in software engineering education. In the following, after introducing a sketch of UtiLisp and its brief history in the second chapter, a few examples of development and improvement, in other words, hacking, will be described.

2. UtiLisp and its Development

UtiLisp, which is not well known outside Japan, is a Lisp dialect of classical MacLisp: eval top loop, shallow binding, dynamic scope rule interpreter, with macro

*Department of Mathematical Engineering, University of Tokyo.

and readmacro facilities.

There are four versions of UtiLisp:

1. UtiLisp360 on main frame machines like Hitac M series, Facom M/S series, IBM 3000 series
2. UtiLisp68 on MC68000 (24 bit address) for Sun 1, Macintosh etc.
3. UtiLisp32 on Mc68010, 68020 (32 bit address) for Sun 2, 3 and on Vax etc.
4. UtiLisp/C on SPARC for Sun 4 and Sparc Station.

According to the survey on Lisp and AI application languages conducted in early 1986, the most widely used Lisps in the Japanese AI community are Franz Lisp (60), then Vax Lisp (40), UtiLisp (33), InterLisp (-D) (33) etc. In this survey, UtiLisp includes all of aforementioned first 3 versions. (Popular AI languages other than Lisp are: C-Prolog (46), Prolog-KABA (44), Smalltalk-80 (38) etc.)

The main reason of the wide use of UtiLisp is its processing speed. Table 1 gives the speed comparison of various Lisps for (tarai 10 5 0) where the function tarai is defied as:

```
(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)))
```

UtiLisp officially stands for the University of Tokyo Interactive Lisp. However, in the beginning UtiLisp was said to be the language for utility programs. Thus, it was called UtiLisp, a portmanteau word of "utility" and "lisp".

The first UtiLisp project was triggered by the Ada system implementation experience and was proposed and designed by Takashi Chikayama, one of the first three students of our laboratory, who is now at ICOT (Institute for New Generation Computer Technology). He is one of the typical talented Lisp hackers and he wrote many Lisp systems. His shiny micro Lisp was designed in 1976 for Intel 8080 microprocessor with 4K byte RAM, and the program size was only 850 byte including garbage collector [1].

In 1979, we read the reference manual of Ada (or Green) Programming Language published in the June issue of ACM Sigplan Notices [2]. We were tempted to write a simple compiler for it, firstly to check the consistency of the description of the manual and secondly to learn about the Ada system. In November and December, 1979, a pilot Ada compiler was being coded by Chikayama with the help of other students for a Hitac 8800 machine at the University Computer Centre [3]. The first phase, the token reader, was written in Pascal 8000 which inputs the Ada source program and outputs the persed tree in the form of nested S-expressions. The remaining phases of the compiler and the run

Table 1 A benchmark of Lisp systems for (tarai 10 5 0) in seconds.

System	Machine	Interpreter	Compiler
UtiLisp360	Hitac M682	3.82	0.233
UtiLisp68	Sord M685	104	11.4
"	NEC PC9801	145	16.2
"	Apple Macintosh	214	—
UtiLisp32	Vax 8600	23.8	2.79
"	Sun 3/260	22.0	2.10
"	Sun 3/50	53.0	4.70
"	Sun 2/120	117	13.5
"	Sord M685	108	—
UtiLisp/C	Sun 4/260	21.07	0.78
"	Sun4/110	31.25	2.08
"	Sparc station	18.17	0.78
"	RISC-NEWS(R3000)	10.77	0.62
Franz Lisp	Vax 8600	81.3	31.6
"	Sun 3/52	262	86.1
"	Sun 2/120	499	159
Symbolics 3640	Symbolics 3640	369	3.85
KCL	Vax 8600	393	8.38

time support routines were all coded in Hlisp which was the only available Lisp in those days for that machine [4]. By the beginning of the following year, the pilot Ada compiler was completed. The time for writing the compiler was short, however the speed of the compiled Ada program was intolerably slow because of the slowness of the Hlisp interpreter.

That experience in Hlisp was enough for us to conclude that Lisp might become a very practical system description language if a super efficient interpreter and ultraclever compiler are designed, since the language already possesses full expandability or the user definable facilities, and its interactive style of programming provides high productivity in program writing.

The computer centre replaced the Hitac 8800 with a Hitac M200H in the summer of 1980. during the replacement, Chikayama designed the new Lisp processor for the Hitac M200H. By the end of 1980, the interpreter of UtiLisp was almost completed. In the spring of 1981, the compiler was coded. After that, he spent all of his time writing a UtiLisp manual [5] and the doctoral dissertation [6][7].

In those days, two other students in our group were endevouring to write programs. One was Hideyuki Nakashima, who was implementing Prolog/KR (KR stands for Knowledge Representation) [8] on UtiLisp, and the other, Michio Kimura, who was developing a medical expert system called Anticipator (ANTIbiotics Counsellor for Infectious PAThogenic Organisms) [9] [10] on Prolog/KR. Those two parallel implementations are quite effective in improving their systems mutually. These pair implementations proved that UtiLisp was a practical system description language.

Then UtiLisp was transferred to Facom main frame machines by members of the Institute of Physical and Chemistry Research and this UtiLisp was subsequently improved by Fujitsu Laboratory. UtiLisp was also transferred to IBM main frame machines (VM and

CMS). This version of UtiLisp is referred to as UtiLisp360 in this article.

Our laboratory installed a Sun 1 Workstation in the spring of 1983. The students naturally wished to have UtiLisp on it. Because Hitac M200 and MC68000 microprocessor of Sun Workstation both have 16 registers 32 bits long and 24 bit address, it seemed relatively simple to move. In the winter from 1983 to 1984, an undergraduate student, Yutaka Tomioka, undertook this operation [11][12]. The main frame UtiLisp was coded in assembly language with fully spangled macro definitions and calls. Unfortunately, the assembly program for MC68000 we used has no macro assembly facilities. Therefore, the first step of transportation was to devise the means to expand macros. This was done in Franz Lisp on Vax on which we developed another UtiLisp interpreter and downloaded it to the Sun Workstation. As will be described later, it worked well. In the case of UtiLisp360, the copying garbage collector was employed because of the large virtual memory space. However, the workstation had only a 1 MB RAM and accordingly, the copying garbage collector was not suitable. So, the ordinary mark and sweep, compactifying garbage collector was designed. The compiler was developed by Hideya Iwasaki in the next spring. This version of UtiLisp was called UtiLisp68. UtiLisp68 was transferred to Sord M685, NEC PC9801, Sharp OA-90 and facom 16β with MC68000 CPU board [13]. It was also transferred to U-station at Shimura Laboratory of Tokyo Institute of Technology.

In the fall of 1985, the UtiLisp68 interpreter was installed on Apple Macintosh with 512 kB RAM except floating point number and reference types by Keiichi Kaneko and Kei Yuasa [14].

Then came the era of Sun 2 and Sun 3 Workstations which have 32 bit data busses. This posed a new problem because we could not use the most significant 8 bits for pointer tags [15]. In the summer of 1986, after considerable time of discussions, a hybrid implementation of pointer tag and object tag was proposed as the best solution. The main transportation was undertaken by Kaneko [16]; the garbage collector was coded by Toshinari Takahashi [17]; the compiler by Shin Ishii [18]. This UtiLisp is now known as UtiLisp32 because of its 32 bit data bus. Until the development of UtiLisp32, UtiLisp didn't have bignum data type. This caused a problem in installing Reduce (a formula manipulation system) on UtiLisp. Bignum should have been implemented much earlier. Thus, Kaneko designed the bignum package for UtiLisp32 [19] and then this system became the first class Lisp.

UtiLisp 32 on Sun 3 was similarly developed using macro functions expanded by user defined Lisp functions. We realized that if the macro definitions are modified properly, the same source code will generate a UtiLisp interpreter for Vaxen. Motivated by this idea, Kaneko wrote a set of macro definitions which will produce a code for Vax. In the course of two weeks, he succeeded in installing a new interpreter for Vax 8600 at the University Computer Centre [20].

In the summer of 1988, the first Sun 4 Workstation arrived. Its heart was the SPARC Integer Unit, entirely different architecture from MC68000. Naturally, our previous UtiLisp's were not runnable on it. However, only a couple of weeks after the arrival, Tetsurou Tanaka tested his Lisp interpreter written in C language on the Sun 4 [21]. He had already designed the basic data structure of UtiLisp/C, (because it is in C language,) and continued to implement the rest part of the interpreter. The data structures were carefully chosen to suit the SPARC architecture though it also might be compiled for other machines of different architecture.

Then an undergraduate student, Masakazu Muramatsu, started his work on a UtiLisp/C compiler which compiles UtiLisp/C functions into C code to be C-compiled and loaded in the later stages [22]. This speeded up the coding time for compilers. This was particularly possible because many primitive functions in UtiLisp/C were coded in C and the coding styles were easy to follow by the compiler.

One of the drawbacks of UtiLisp/C written in C language was the routine set for bignum arithmetic. Programs in C language are not so flexible to treat the carry propagation. In the spring holidays of 1988, the author wrote bignum arithmetic in SPARC assembly language, partly to learn the instruction set and the basic architecture of the SPARC chip [23].

As described above, many students contributed in improving a series of versions of UtiLisp. Although not mentioned explicitly in the previous paragraphs, many more students improved systems partially or gave good suggestions or cooperated with their colleagues in trouble shooting.

When we were requested, we shipped UtiLisp system to any computer center. Some of these centers are eager users of the system; they send us bug information they found. They helped us in fixing bugs of the system too. UtiLisp systems have been developed by NEC independently. They installed UtiLisp on Acos 4 and Acos 6 (their main frame machines). When NEC released an Engineering Workstation EWS 4800, they implemented a UtiLisp MC68020 version with their own scheme [24].

When an expert shell which was installed on UtiLisp is exported, UtiLisp is also exported with the expert system to customers abroad.

3. UtiLisp for Hitac M200H—UtiLisp360

As already introduced, UtiLisp was developed by Chikayama on a Hitac M series machine with VOS3 operating system. UtiLisp was designed not only for symbol manipulation but also for system description. Accordingly, it is highly tuned up for character and bit string manipulation without sacrificing the flexibility and expandability of original Lisp. For flexibility, it has a set of functions for use with the operating system facilities. Users may have access to the key data structure of the system. For instance, users may see the stack, know the address of symbolic atom information, modify read table, may create his own obvector (=oblist in vector form), may check the number of arguments for machine code functions etc.

Data types of UtiLisp include: cons cell, symbolic atom, fixnum (=fixed point number), flonum (=floating point number), (in the later versions, bignum (=multiple precision fixed point number,)) string, input/output stream, vector, reference (=vector element), and code piece (=function in machine language). These types are discriminated by pointer tags like Cambridge Lisp [25]. Type discrimination was not accelerated by solely employing pointer tags. Tag values are arranged in sophisticated fashion so that the most basic discrimination would be performed quickly.

Strings are implemented in packed form to utilize the hardware functions. The elements of string are characters; however by character we mean a short fixnum from 0 to 255. So, all character positions of the 8 bit code table are treated uniformly as characters. The string type has a set of bit manipulating functions.

UtiLisp has one dimensional vectors. Accessing an element of a vector is quicker than accessing an element of a list data, although the necessary size must be known in advance for the vector.

In UtiLisp, you are not allowed to write functions, whose parameter will not be evaluated like fsubr in Lisp 1.5. Special forms only exist in built-in functions. When it becomes necessary for a user to write special forms, he is recommended to use macros. The macros of UtiLisp are similar to those of MacLisp, a list of unevaluated parameters is sent to the macro with one parameter and the body is evaluated, then the result is evaluated again. The macro speed might be mistaken to be slow. However, the compiled macro has no overhead at all.

Default parameters are so convenient to use. If an element of a lambda list is a list whose car is a symbolic atom, then if the corresponding actual parameter is not supplied, the cdr of the list is evaluated from left one after another and the final result of evaluation is bound to the symbolic atom as a default value. Default parameters may be implemented by other means. However, this default parameter system is quite readable.

During evaluations, several errors may take place. UtiLisp has a set of default error handling functions. For instance, standard function `err:end-of-file` is stored in the variable `err:end-of-file`. The user may, however, replace the error functions at will if he wishes to. As an example, if he is to read in a file and finish the reading gracefully at the end of file, he stores the standard end-of-file error function in his temporary variable, sets his own error function instead of the standard one. Then at the end of file, his function gets the control; it restores the end-of-file function with the standard one and he may return to the master program.

A stream is a set of standard functions and data areas. In a sense, it is a kind of package; the idea of i/o stream was borrowed from OS6 designed by Strachey and Stoy of Oxford University [26][27].

In implementing the main frame UtiLisp system, macro assembly facilities were used to a great extent. Nevertheless, the list is not so difficult to read. As a simple example, the macro "iflist" is shown. This sequence checks whether the tag in a special register &r is negative or not.

```
    macro
&l  iflist &r, &adr if &r is a list then go to &adr
&l  ltr    &r, &r
    bnh    &adr
    mend
```

Combinations of car's and cdr's are expanded by macros, and as a natural extension, UtiLisp has function `cr` as the least element of cars' and cdrs' family.

UtiLisp has its own editor called Use (UtiLisp Structure Editor) and a pretty printer called Prind. Use is almost similar to the structure editor of InterLisp. Prind is a clever pretty printer. It folds back a backquote read macro facility. For example, `(list. 'setq (list . 'plus x n))` will be printed as `` `(setq ,x (plus ,x ,n))) ``.

4. UtiLisp on MC68000—UtiLisp68

The first Sun Workstation came with a simple monitor implemented in its ROM. Therefore, we had to develop many system programs ourselves. We already know that UtiLisp is a general-purpose language, so we began to develop a UtiLisp system on MC68000 with the top priority. To transfer UtiLisp means, conceptually, to transfer that speed and flexibility and that excellent human interface and, technically, to copy the precise structure and functionality of main frame UtiLisp as close as possible. Functionality is represented in the form of macro definitions. So the necessary work was to invent mechanisms to expand macros of the main frame computer for MC68000 assembly programs and to define each macro in terms of the MC68000 machine instructions.

Our solution was to write a system like LAP (Lisp assembly program) of Lisp 1.5 where all machine instructions, macro calls, pseudo instructions (=directives) in the form of S-expressions. Then, any Lisp program may treat the source program freely. A 68000 assembly program has its unique instruction format especially for address modes. We need to express that in a simple list format. Our orthography is:

```
Rn          —>Rn       data and address register direct
an@         —>an@      address register indirect
an@+        —>an@+     ditto with postincrement
an@−        —>an@−     ditto with predecrement
an@(d)      —>(an@ d)  ditto with displacement
an@(d, Ri: W) —> (an@ d Ri W) ditto with index
normal      —>normal   absolute address
#immediate —>'immediate immediate data
```

Labels are in the form of symbolic atoms just like those in program feature. Comments are preceded by semicolon like Lisp programs. The lap thus developed is called lap68. In lap68, "iflist" macro is defined as:

```
(dm iflist (an addr)
  (lapl '((movl, and d0)
          (jmi, addr)))))
```

Function dm defines macro iflist and, at the same time, places the indicator in the property list of iflist saying that this is a lap68 macro. Function lapl expands each element of the list given as the parameter. In case of a complicated macro definition like that of the car-cdr expander, the macro is defined as:

```
(dm cxr (fname)
  (lapl '(subr ,fname (x)))
  (setq fname (string fname))
  (do ((index (/- (string-length fname) 2)
        (/1- index)))
    ((/0= index)(lapl '((return0)(codend))))
    (lapl  (cond  ((=(sref  fname  index)
                   (character ''a''))
                   '(cara))
                  (t '(cdra))))))
(dm cara nil
  (lapl '((ifatom A typerr)(movl (A@ car) A))))
(dm cdra nil
  (lapl '((ifatom A typerr)(movl A@ A))))
```

Here, lap1 is a lap function to be used when the argument list contains single element. Other functions shown here are used without definition. Now, car-cdr composite functions are defined as:

```
(cxr cr)
(cxr car)
. . .
(cxr cddddr) etc.
```

The first version was written in Franz Lisp. However once the first UtiLisp was completed, the lap and source were converted to UtiLisp and the production system was moved to Sord M685 at our laboratory (CPU of Sord M685 is MC68000). Debugging and productions were repeated on this version.

The garbage collector of UtiLisp68 is mark and sweep, compactifying one. To mark all reachable objects, the pointer reverse method, i.e., Deutsch, Schorr and Waite algorithm, is used [28]. Compaction is based on Morris' algorithm [29]. However, applying those algorithm was not simple. First, there are various size of objects in a heap area; binary cons cells, symbolic atom objects, various length vector and string cells. Machine code pieces are allocated in a different area because these objects seldom become garbages.

Marking a reference pointer is performed as follows. A reference pointer points to an element of a vector. First, the marker traces up the vector body to find the header. If it is marked, descendant marking is finished. If not, the pointed element pointed is marked with "stop" bit in tag field, and the element immediate above the pointed one is marked, i.e., the pointer is reversed to point to the parent object. Then elements are marked upward one after another and when marking reaches the header, using the information about the length of the vector object contained in the header, the marker goes down to the bottom element. By repeating this it finally reaches the element marked with stop. After marking that one, marking of that reference pointer ends.

When Morris' algorithm is used, the whole heap area should be scanned twice, once downward then upward. For downward scanning, the large objects have special header object tags; so its whole body is skipped based on the size information stored in the header. For binary cons cell, the sweeper skips two; as the size of the symbolic atom object is four, the third word is also marked as non garbage, therefore it is skipped by two words twice. So it is all right to sweep down. However, since there is no header information at the bottom of the objects, while sweeping downward, the contents of the header and the bottom word are swapped. And moreover, while sweeping the continuous garbages, the sweeper remembers the last non garbage word; when it finds the next non garbage word, it places so called upward garbage skip links so that in upward sweeping, garbages are skipped in one action. Of course, the swapped contents should be restored.

5. UtiLisp for MC68020—UtiLisp32

When UtiLisp was first implemented for Hitac M series machines and then transferred to MC68000, since those architectures use only 24 bit address, top 8 bits were used for tags with which type discrimination was accelerated. However, with the newest microprocessor, MC68020, as it uses all 32 bits for address, UtiLisp68 could not be directly transferred to the new machine. So, we changed the object representations slightly. Speed of UtiLisp was gained by its quick type checking based on the tag information. In the new implementation, types should be discriminated quickly. In this new implementation, types are represented as follows. The size of fixnum is limited to 28 bits. The value is shifted 2 bits to the left; the least significant 2 bits are for garbage collection marking. The most significant 2 bits are set to 11, thus the representation of fixnum is seen as negative. Next, the heap address is assumed as positive, i.e. the address is less than 0x40000000. The heap area is divided in three, that is:

- for symbols,
- for other atoms (or others for short) and
- for cons cells

and located in that order placing the symbol area in the smallest address. (In "other atoms" area, vector, string, stream, head part of code pieces, flonum and bignum objects are placed.) Two address registers are provided for type checking; one is nil address which points to the symbolic cell for nil that is placed at the largest address of the symbolic area. The other is a pointer which always points to the boundary of cons cell area which grows to the smaller address direction. This pointer is called list top in the next paragraph.

So, if this arrangement is seen as unsigned, the smallest is symbols, then nil, others, cons cells, fixnums. Now if it is seen as signed, then the smallest is fixnum, then symbol, others, and cons cells. Therefore, atoms are smaller than list top in signed; lists are larger than list top in signed. Symbols are smaller than or equal nil pointer unsigned; fixnum is negative or smaller than 0xc0000000 signed.

Each object placed in the others area has an extra 32 bit header which has an object tag beginning with 10. The header also contains the length of the object. When a pointer points into the others area and the object pointed to has object header, then the type is known by that header information. If the pointed object has no header tag, then the pointer was reference type which points to the vector element. Thus, the discrimination is slowed down for special objects located in the others area. But for other genuine Lisp objects, speed is maintained as in the former implementation.

Another remarkable story about UtiLisp32 is a bignum print routine. As introduced earlier, UtiLisp32 included a bignum package. When many bingum benchmark programs are evaluated and cpu times were compared between UtiLisp32 and Franz Lisp on Vax, Franz Lisp won in printing big numbers. The reason was that Vax has no unsigned division necessary for bignum conversion represented as the sequence of 32 bits long words. The Franz Lisp representation is a series of 31 bits words. So, a big number could be converted by dividing larger powers of ten with Franz Lisp than with UtiLisp32. One student, Takahasi, tried to improve the conversion. Finally he invented the conversion methods in which a big number is divided by 10^{12} (or more precisely, by $5^{12}*2^4$), thus gaining the conversion speed. This is too tricky yet it works perfactly [30].

6. Multiprocess UtiLisp—mUtiLisp

We are interested in a multi-processing version of UtiLisp which is called "mUtiLisp" [31]. It was designed and implemented by Iwasaki. In mUtiLisp, a process may create its child process with the function:

```
(fork process-name . body)
```

Interprocess communication or synchronization are made through the message passing with:

```
(send process message)
```

and

```
(receive process) or (receive)
```

A child process inherits the environment from its parent. A process may be stopped with the functions:

```
(suspend) or (suspend process).
```

Or it may be restarted with the function:

```
(resume process).
```

A simple example is a prime number generator as shown below:

```
(defun sift nil
  (lets ((prime (cdr (receive)))
         (n)(x)(next))
    (cond
      (prime
        (print prime)
        (setq next (fork (genprocname)(sift)))
        (setq n  (+ prime prime))
        (loop
          (setq x  (cdr (receive)))
          (cond ((null x)(send next nil)(exit)))
          (do nil
            ((<=  xn)(and(<  xn)(sendnextx)))
            (incr n prime))))))
(defun prime (to)
  (do ((next (fork (genprocname)(sift)))
     (i 2 (1+ i)))
     ((<= to i)(send next nil))
     (send next i)))
```

On this multi-process kernel in UtiLisp, a prototype operating system is being implemented. The preliminary operating system was once designed by an undergraduate student, Nishio, with a Unix like file system [33]. The file system was in the form of the nested S-expression with files at its terminal nodes; the file system was installed in a large heap area.

Work on multi-process UtiLisp are still in progress.

The interpreter is being implemented on the workstation with multiprocessors. The single processor version of mUtiLisp was evaluated by Minoru Terada through three experiments, i.e., operating system described in mUtiLisp, the multiprocess GHC interpreter and the remote evaluator over the mUtiLisp system coupled via network [34].

7. UtiLisp for SPARC—UtiLisp/C

UtiLisp system should also be implemented for the new machines with SPARC cpu. The implementation was undertaken by Tanaka, and even before the machine arrival, he weighed the alternative implementation techniques and decided to write the UtiLisp iterpreter in the C language for the first time. The main reasons were:

- The architecture was quite new for us to write correct and efficient code in assembly language.
- The optimizing C compiler will generate good code.
- The interpreter will be transferable.
- The correctness of the interpreter will be easily checked.
- Debugging will also be easy.

Thus the interpreter is called UtiLisp/C.

Of course, many architectural facilities were fully used in the program for instance, the tagged arithmetic were employed where appropriate. Tagged arithmetic was included in the code by means of macro expansion.

Data structures were redesigned for the new machine. Since most date types are multiple of four bytes, the least two bits are assigned as tags. As in the case of UtiLisp32, the principal data types are cons'es, symbols and fixed point numbers (fixnums). Flonums, bignums, vectors, strings, streams and code are grouped as "others". Others is the fourth main data type.

Tag bit combinations correspond to the four main data types as:

- 00 fixnum
- 01 symbol
- 10 cons
- 11 others

Types of others group listed above have data which point to object tags that further classify the data types. One exception is the data type "reference" which is simply the pointer to the vector elements. No Lisp object corresponds to reference data type. References are identified by the fact that it has "others" tag and the pointed location is not the place for object tags.

The two bit tags are so designed that the word boundary error may catch the data type error. In other words: in case of fixnum, tagged arithmetic will detect addition or subtraction on non fixnum operands; in case of cons, the pointer refers the second byte of car part and the basic operation "car" adds offset -2 to the pointer, "cdr" adds offset $+2$ to the pointer. If the operand is the genuine cons type data, the resultant values correctly indicate the word boundaries. Otherwise, e.g., in case of symbols, the resultant values indicate inside word boundaries, generating bus error traps.

UtiLisp/C adopted copying garbage collection.

The UtiLisp/C compiler is unique too in the series of UtiLisp compilers because it first generates the object program in C language. Until the previous implementation, i.e., upto UtiLisp32, compilers generated object code in assembly language in the computer, local assembly programs produced the object in machine code, and the object then was linked. In case of UtiLisp/C compiler, it compiles Lisp functions to C code, evokes C compile to compile the generated programs, then "incrementally loads" the compiled file into a Lisp process. Finally the Lisp object of code type is created.

The decision criteria for C code generation are:

- Transference to other machines is simple.
- Good optimizing C compiler is available.
- It is amicable to an interpreter in C language.

However, compilation is apparently slower since many operations must be executed. As the purpose of compilation is to obtain a faster version of programs, it seems that the compiler's mission is fulfilled when efficient object programs are generated though the compiling speed is a little bit slow.

8. Conclusion

Software development seems to be a complicated social and psychological phenomena where many factors contribute or interact. In the case described so far, the first corner stone was an idea of one of the students to implement an efficient Lisp interpreter. This nucleus indeed worked so well that it kept interest of the newcomer students. They in fact improved the system in various sections. Concrete examples were presented in each of the sections. Although the number of examples is limited, in the real system, there appeared many interesting ideas. And all in all, the systems presented such high efficiency without sacrificing flexicility of the Lisp language.

As the result of almost ten years' discussion, coding, retrospect and maintenance, five UtiLips systems came to exist. (UtiLisp68 is now absorbed by UtiLisp32 because of the compatibility.) They are real software products. However, inside the laboratory, rather autonomous production process only motivated by the pure spirit to produce faster system seems, at least to the author, more interesting.

References

1. CHIKAYAMA, T. Micro Lisp, *Proceedings of Programming Symposium on Microcomputer Software* (July 1977), 96-100 (in Japanese).
2. Preliminary Ada Reference Manual, Sigplan Notices, **14**, 6 (June 1979), Part A.
3. CHIKAYAMA, T. A Pilot Compiler for Programming Language Ada, *Proceedings of the 21st Programming Symposium*, IPS Japan (Jan. 1980) (in Japanese), 137-142.

4. Kanada, Y. HLisp and Supplementary Hlisp-Reduce Manual, appendix to the Doctoral Dissertation titled "New Algorithms for Symbolic Formula Manipulation" (Feb. 1979).

5. Chikayama, T. UtiLisp Manual, Technical Reports METR 81-6, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, Univ. of Tokyo (Sept. 1981).

6. Chikayama, T. Study of Implementation of Lisp System, *Doctoral Thesis, Information Engineering Course, Univ. of Tokyo* (March 1982) (in Japanese).

7. Chikayama, T. Implementation of the UtiLisp system, *Trans. IPS Japan*, IPS Japan, **24**, 5 (1983) (in Japanese) 599-604.

8. Nakashima, H. Prolog/KR User's Manual, Technical Reports METR 82-4, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, Univ. of Tokyo (1982).

9. Kimura, M., Koyama, T., Kaihara, S. and Tsuchiya, F. Anticipator: A Medical Expert System Implemented by Prolog/KR, *J. IPS Japan*, **7**, 3 (1984), 149-156.

10. Kimura, M. Construction of an Antibiotic Medication Counselling System by Means of Knowledge Engineering, Technical Reports METR 83-1, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, Univ. of Tokyo (Sept. 1983).

11. Tomioka, Y. Macros for System Program Description and Transportation of Programming Language System, Bachelor's thesis, Department of Mathematical Engineering and Instrumentation Physics (March 1984).

12. Wada, E. and Tomioka, Y. Transportation of UtiLisp to 68000, *Preprints of WGSYM Meeting*, IPS Japan (Oct. 1984) (in Japanese), 15-21.

13. Terada, M. and Wada, E. Transportation of Object Files between the Systems with Common CPU, *Preprints of SIGSW Meeting*, IPS Japan, **87**, 11 (Feb. 1987) (in Japanese), 89-95.

14. Yuasa, K. and Kaneko, K. Transportation of UtiLisp to Macintosh, the Days of Hardship, *Proceedings of the 27th Programming Symposium*, IPS Japan (Jan. 1986) (in Japanese), 131-141.

15. Yuasa, K. A New Scheme of Object Allocation for Lisp Systems, *Trans. IPS Japan*, IPS Japan, **30**, 7 (July 1989) (in Japanese), 849-855.

16. Kaneko, K. and Yuasa, K. A New Implementation Technique for the UtiLisp System, *Proprints of WGSTM Meeting*, IPS Japan, **41**, 7 (Jun. 1987).

17. Takahashi, T. Garbage Collection for a Complex-Tagged Lisp Processor and the Related Issues, *Trans. IPS Japan*, IPS Japan, **30**, 3 (March 1989) (in Japanese).

18. Ishii, M. Compiler Construction for Symbol Processing Languages, Master's Thesis, Information Engineering Course, Univ. of Tokyo (March 1988) (in Japanese).

19. Kaneko, K. Numerical Operations in Nonnumerical Languages, Master's Thesis, Information Engineering Course, Univ. of Tokyo (March 1987).

20. Kaneko, K. UtiLisp on Vax, Computer Centre News, **19**, 2, 1987, Computer Centre, Univ. of Tokyo (in Japanese).

21. Tanaka, T. The UtiLisp/C Interpreter, *Preprints of WGSYM Meeting*, IPS Japan, **53**, 3 (Nov. 1989) (in Japanese).

22. Muramatsu, M. The UtiLisp/C Compiler, *Preprints of WGSYM Meeting*, IPS Japan, **53**, 5 (Nov. 1989) (in Japanese).

23. Wada, E. Bignum Routine for UtiLisp/C, *Preprints of WGSYM Meeting*, IPS Japan, **53**, 4 (Nov. 1989) (in Japanese).

24. Hashimoto, Y., Uchida, S., Ogawa, Y. and Takahashi, T. Implementation of Lisp on 32-bit-addressing Chips, *Preprints of WGSYM Meeting*, IPS Japan, **41**, 6 (Jun. 1987) (in Japanese).

25. Fitch, J. P. and Norman, A. C. Implementing LISP in a High-level Language, Software-Practice and Experience, **7**, 6 (1977), 713-725.

26. Stoy, J. E. and Strachey, C. OS6-An Experimental Operating System for a Small Computer. Part 1: General Principles and Structure, *Computer Journal*, **15**, 2 (1972), 117-124.

27. ibid. Part 2: Input/Output and Filing System, *Computer Journal*, **15**, 3 (1972), 195-203.

28. Knuth, D. E. Fundamental Algorithms, The Art of Computer Programming (Vol. 1), p. 417, Addison Wesley (1973).

29. Morris, F. L. A Time- and Space-Efficient Garbage Compaction Algorithm, CACM, **21**, 8 (Aug. 1978), 662-665.

30. Takahashi, T. Algorithms to Indicate Multiple-Precision Integer with Decimal Notation, *Trans. IPS Japan*, IPS Japan, **30**, 9 (Sept. 1989) (in Japanese).

31. Iwasaki, H., Terada, M. and Higuchi, H. Parallel Lisp System and its Application, *Proceedings of the 28th Programming Symposium*, IPS Japan (Jan. 1987) (in Japanese), 131-141.

32. Terada, M. Experimental Environment in Multi-Processing OS "SOKO", *Trans. IPS Japan*, IPS Japan, **30**, 3 (March 1989) (in Japanese).

33. Nishio, N. Lisp-based Operating System, Bachelor's thesis, Department of Mathematical Engineering and Instrumentation Physics (March 1986) (in Japanese).

34. Terada, M. System Description by Parallel Lisp and Its Evaluation, Doctoral Thesis, Information Engineering Course, Univ. of Tokyo, Jan. 1990 (in Japanese).

(Received February 22, 1990)