SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(10)). 1015–1026 (OCTOBER 1991)

Floating-point Number LISP

KYOJI UMEMURA
Software Laboratories, Nippon Telegraph and Telephone Corporation, Room 5-113A, NTT Laboratories, 3-9-11 Midori-cho, Musashino, Tokyo 180, Japan

SUMMARY

A floating-point-number-oriented LISP has been developed. Though it performs type checking on every numeric operation, it runs as fast as Fortran for simple differential-equation problems. This paper describes the implementation, provides some measurements of its efficiency, and discusses the feasibility of this type of implementation.

KEY WORDS Floating-point arithmetic LISP Symbolic computation

INTRODUCTION

There exist many applications where both symbolic and floating-point computations are required at the same time. Image recognition, for example, requires huge amounts of numeric calculation to extract features from raw image data before the actual recognition is performed according to those features.

Systems for symbolic computation should perform type-checking on numeric calculations. Although the overhead can be eliminated by declarations to the compiler, our observation shows that it is not always easy to make the appropriate declarations. The overhead can be eliminated by special hardware, but use of special machines sometimes makes system development difficult.

This paper describes a LISP system, suitable for both symbolic and numeric computation, which has been developed under VAX/VMS. It runs simple differential-equation problems nearly as fast as VAX Fortran with no declarations.

This system, called Real Number Lisp, benefits people who need both the flexibility of a symbolic computation system and the capability of a numeric computation system at the same time. The same techniques can be used with other symbolic computing languages, where dynamic type-checking is required.

DATA REPRESENTATION

In Real Number Lisp, all numbers are represented in double-precision format. Both the integer 1 and the floating point number 1·0 are represented in the same way. Rounding errors have no effect on fixed-length integer operations because add, subtract and multiply operations in double-precision numbers are accurate within a 56-bit range on DEC VAX computers.

All other data are represented as invalid floating-point numbers. Invalid numbers

0038–0644/91/101015–12$06.00 *Received 5 April 1989*
 Revised 13 November 1989 and 3 April 1991

have a negative sign and a zero exponent field on the VAX computer. They may have any bit pattern in their fraction field. There are 2^{56} invalid double-precision numbers, as there are 56 bits in the fraction field. This is sufficient to represent the other data.

Data-type information is held in the first long word (32 bits). This word contains a sign field, an exponent field and a 24-bit fraction field when used for number representation. The type indicator, often called the 'tag', is placed in this field for non-number representation. The 24-bit width is sufficient for data-type discrimination.

Data-address information is contained in the second long word. This word has only a fraction field for number representation, and contains the data address for link representation. A 32-bit word is sufficient for the VAX address space. Figure 1 explains the representation.

Memory-management fields used in garbage collection are not included in type fields. They are implemented in a separate bit table, because double-precision numbers have no extra space for a marker indicating whether this unit of memory is being used or not.

All data are represented by 64 bits. This is double the number of bits required by most major implementations. Non-numeric data are represented by double-length words, and numbers are represented in the same format as Fortran double-precision numbers.

HARDWARE EXCEPTIONS

Modern computers have circuits or microcode for type checking. The VAX computer checks the data format on every arithmetic operation, and signals an exception when it finds a negative sign and zero exponent data.

When a program tries to perform an arithmetic operation on non-numeric data, such as CONS cell data, Real Number Lisp is able to detect this error without any instructions. All non-numeric data are represented in an invalid floating-point number format, so that arithmetic operations on them mean arithmetic operations on invalid floating-point numbers. It is the arithmetic unit rather than user-written code that detects this situation and signals an exception.

It is thus possible to use the existing hardware exception mechanism for type checking. This can reduce the overhead on dynamic type-checking for arithmetic operations. The code generated for arithmetic computation is similar to Fortran

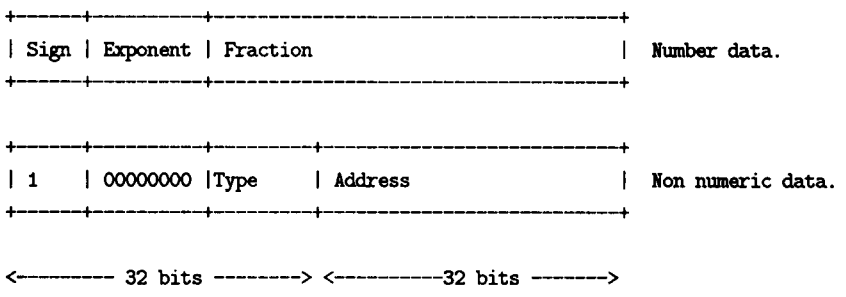

Figure 1. Data representation

code. Although there is no improvement for code that expresses non-numeric operations, no disadvantages exist for such operations.

COMPILED-CODE EXAMPLES

Some code generated by the compiler is presented here, since examining compiled code is one of the best ways to understand the whole system.

This compiler generates VAX instructions. The VAX is a 32-bit machine which can handle 8 bits (byte), 16 bits (word), 32 bits (long word), 64-bit integers (quadruple word) and 64-bit floating-point numbers (double precision) with single instructions. Most instruction mnemonics have an extra character for representing the format of the data being handled. The instruction MOVD X,Y, for example, means 'MOVe Double'. It moves double precision data from X to Y and signals an error if X contains an invalid floating-point number. The instruction ADDL3 A, B, C means 'ADD Long words and has 3 operands'. It adds long words A and B and stores the result in C. All registers are 32 bits long. Two successive numbered registers are used for a quadruple word or double-precision number. The instruction MOVQ X,RO, for example, modifies R0 and R1. Operands can be registers, immediate data or data in memory pointed to by a register. For example, R0 is register zero, #8 is an immediate value. 8(R10) is the memory at address 8+R10; and @8(R10) is the data pointed to by 8(R10). See Reference 1 for more details on VAX instructions.

Registers are used in a special way. Arguments are passed by R0&R1 and R2&R3. Local variables are contained in R4&R5, R6&R7, R8&R9. Register R10 is used as a data stack pointer and R11 is used as a scratch address. The argument pointer AP in Real Number Lisp points to the system global data. 16(AP) means the eighth system global address. For example, 8(AP) contains the tag pattern for the data type CELL; -4(AP) contains the address of the function ILLEGAL-CAR-CDR; and 96(AP) contains the third argument. The argument pointer AP is not used for arguments because the compiler generates neither CALLS nor CALLG instructions which use AP as an argument pointer.

The first example (Figure 2) is one of the simplest arithmetic functions. This function is converted into two instructions. Although there is no code for type-checking, it signals an error if either of the arguments is not a number.

The second example (Figure 3) is one of the simplest functions of any list operation. This time, type-checking code is generated, as in most implementations. Although 64 bits of data are loaded into registers, one instruction is enough for this operation.

The third example (Figure 4) is a simple list function that gets the CDR part of the last cell of a list. The instructions marked '*' are not necessary. This compiler does not use type-inference methods; it does not have any access to the information that the variable x must always hold list data as the result of a previous branch.

The last example (Figure 5) is a simple iterative function using floating-point

```
(defun plus (x y) (+ x y))
.....................................................................
L56:    ADDD2 R2 ,R0            ; RO&R1 &= RO&R1 + R2&R3 (double precision)
        RSB
```

Figure 2. Simple arithmetic function

```
(defun rest (x) (cdr x))
.....................................................................
L85 :   ADDL3 #8, R1,R11        ; get address of CDR on R11.
        CMPL    R0,8(AP)        ; check that RO is CELL.
        BEQL    L78             ; goto L78 if R0&R1 contains CELL data.
        JSB     0-4(AP)         ; signal error if not CELL.
L78 :   MOVQ    (R11) ,R0       ; R0&R1 := MEMORY[ R11].
        RSB                     ; return to caller.
```

Figure 3. Simple list function

```
(defun fn (x) (if (atom x) x (fn (cdr x))))
.....................................................................,
L65:    CWL     R0,8(AP)        ; R0 contains the data type information.
        BSQL    L72             ; goto L72 if argument is CELL type.
        RSB                     ; return value is already on R0 end R1.
                                ;return to caller.
L72 :   ADDL3   #B, R1,R11      ; R1 contains address of CELL.
        CMPL    R0,8(AP)        ; make sure that R0&R1 is CELL.          *
        BSQL    L85             ; branch on normal case.                 *
        JSB     IQ-4(AP)        ; signals error. (resume if R11 = NIL) *
L85 :   MDVQ    (R11) ,R0       ; R11 contains address of CDR part.
        MOVL    $FN+4,R11       ; move symbol address to R11.
        JMP     @24(R11)        ; cell func FN. argument is in RO&R1.
                                ; offset 24 means 'Call with one argument. '
                                ; tail merge optimization.
```

Figure 4. CDR of last list cell

numbers. Four instructions are executed in each loop. The compiler does not optimize the location of code; three instructions could do the same function if the code setting the returned value were moved after the end of the loop (thus eliminating a branch).

Though the current compiler could be optimized further, the generated code is still compact. The actual code for arithmetic operations looks like Fortran code.

SPEED MEASUREMENT

Before discussing the feasibility of this implementation, some measurements are necessary for evidence. Measurement was done on a VAX8650, which is one of the larger VAX processors, and which has good floating-point arithmetic units. The executing process has 1 megabyte of real memory, which is enough for execution of the programs used. Measurements were also performed on a VAX780 and on a VAX750, on which many programs have been benchmarked.

All programs here were tested with no type declarations and no compiler options. The compiler generates safe code, which performs type-checking and range-checking.

```
(defun floop (n)
   (let ((x 1.0) (d (/ 1.0 n)))
        (loop (where (< x 0.0) (return 'ok))
              (setq x (- x d)))))
.......................... ..........................................
L65 :    MOVQ    R4, (R10)        ; save R4%R5 on data stack.
         HDVQ    R6,8(R10)        ; save R6&R7 on data stack.
         HOVQ    R8,16(R10)       ; save R8&R9 on data stack.
         SUBL2 #24, R10           ; increment data stack pointer. (note*1)
         HOVQ    R0 , R4          ; move argument to R4&R5. (binding)
         MOVQ    #8, R0           ; RO&R1 := 1.0.
         DIVD3   R4,#8, R2        ; R2&R3 := 1.0/N              (note*2)
         MOVQ R0 , R6             ; R6&R7 is local variable(x).
         MOVQ R2 , R8             ; R8&R9 is local variable(d).
L95 :    TSTD R6                  ; test variable (x).                    *
         BGEQ    L05              ; goto L05, if x >= 0.0.                *
         MOVQ $0K ,R0             ; R0: = Symbol tag, R1:= address
         BRB     L10              ; goto exit sequence.
L05 :    SUBD2 R8, R6             ; x := x-d.                             *
         BRB     L95              ; loop again                            *
L10 :    ADDL2   #24, R10         ; decrement data stack pointer.
         MOVQ    (R10) ,R4        ; recover registers R4 and R5.
         MOVQ    8(R10) ,R6       ; recover registers R6 and R7.
         MOVQ    16(R10) ,R8      ; recover registers R8 and R9.
         RSB                      ; return to caller.

note. (*1) Stack Overflow is detect ad using memory protection mechanism:
          one of MOVQ instructions above may signal an exception.
          The stack pointer, R10, is incremented after saving data
          in order to make recovery consistent. As R10 is not used
          as stack other than Lisp, increment operation can be delayed.

note. (*2) DIVD3 A, B, C performs C := B / A
```

Figure 5. Loop using floating-point number

The worst case

The first program is the eight queens puzzle, putting eight queens on a chessboard such that none of them can take any other. The program and the measurements are shown in Figure 6.

This problem is the worst case for the present implementation. It is necessary to handle double the information of most other LISP implementations because list data are represented in 64 bits rather than 32 bits.

```
;;;
(defvar  *count*)
(defun queen () (setq *count* 0) (qu$ 8 0 0 () () ()) *count*)

(defun qu$ (n i j col up down)
  (cond ((= i n) (incf *count*) ())
        ((= j n) ())
        ((or (qu-hit j co1)
             (qu-hit (+ i j) up)
             (qu-hit (- i j) down))
         (qu$ n i (+ 1 j) col up down))
        ((qu$ n
              (i+ i)
              0
              (cons j col)
              (cons (+ i j) up)
              (cone (- i j) down)))
       ((qu$ n i (+ 1 j) col up down))))

(defun qu-hit (n l)
   (cond ((null l) l)
         ((= n (car l)) l)
         (t (qu-hit n (cdr l)))))
```

VAX8850/Real Number Lisp	0.7 (See)
IBM RT/PC Lucid C-n Lisp	0.97
SUN3/180 Lucid Common Lisp	1.34
VAX780/Real Number Lisp	5.0
VAX750/Real Number Lisp	8.2

Figure 6. Eight queens program and measurement

An intermediate case

The second program is the fast Fourier transformation program, involving memory references and floating-point calculations (see Figure 7). This implementation uses floating-point calculation even though integer calculation may be adequate for a typical user. This program was also measured on several computers; see Reference 2.

A general vector is used where a vector of numbers is adequate, because Real Number Lisp currently does not support arrays of specific type. The array-reference code is slower than Fortran because it does a range check for safety, and converts the indexes into integers. Array-indexing is one of the weakest parts of this implementation.

The best case

The last program is a rather basic differential-equation solution involving a simple loop using floating-point numbers. It does only floating-number calculations in every step within the loop part.

This program runs nearly as fast as Fortran on the same machine. The main difference is due to the final step of the Fortran compiler's optimization. Although optimization of the current compiler is not as good as that for VAX Fortran, it would be possible to generate the same code as VAX Fortran if the optimizer were improved further. The programs and measurements are shown in Figure 8.

LANGUAGE RESTRICTION

This implementation cannot satisfy the Common LISP number specification. [3] Its numbers are restricted to floating-point numbers, i.e. large numbers and complex numbers cannot be implemented in a sound way.

This restriction is fatal for some LISP applications. Formula processing cannot be executed without arbitrary-precision integers, and some applications of this kind therefore cannot use our system. However, some programs can be executed without modification. In that case, the user can enjoy safety and efficiency at the same time. This feature is important, for example, in interactive analysis of numeric data.

FLOATING-POINT REGISTERS

The compiler uses the fact (specific to VAX architecture) that the VAX registers can be used for both floating-point and integer operations. If separate registers existed as they do on many machines, instructions for numeric computation would move data into a floating-point register and codes for symbolic computation would move data into two integer registers. This difference is similar to the difference between C compilers for the VAX and C compilers for other processors. Fortunately it does not affect the technique described here.

MEMORY USE

This implementation requires twice as much memory for data as most other implementations because it uses 64 bits for each item whereas most others use 32 bits.

Real Number Lisp carefully allocates data on quadruple-word boundaries. If the physical memory bus is at least 64 bits wide, both 32-bit data and 64-bit data can be obtained in one memory cycle. Therefore, there is no reason that `MOVQ (R11 ), R0` should be slower than `MOVL (R11 ),RO`.

Introducing type-specific structure is the right way to solve the memory problem. Although it is true that this implementation requires twice the real memory of most other implementations for list processing, modern LISP programs have come to use structures rather than lists. A type-specific structure can reduce memory requirements. Furthermore, since computers are continually being produced with more and more real memory, memory requirements are becoming a less significant problem.

```
;;; This is an FFT benchmark written by Harry Barrow
;;; General vector is used.
;;;
(defvar re (make-sequence 'vector 1025))
(dotimess (i 1025) (setf (svref re i) 0.0))
(defvar im (make-sequence 'vector 1025))
(dotimes (i 1025) (setf (svref im i) 0.0))
(defun fft (areal aimag)
   (prog
        (ar ai i j k m n l e le1 ip nv2 nm1 ur ui wr wi tr ti)
        (setq ar areal
              ai aimag
              n (length ar)
              n (- n 1)
              nv2 (floor n 2)
              nm1 (- n 1)       ; (*1)
              n 0
              i 1)
l1      (cond ((< i n) (setq m (+ m 1) i (+ i i)) (go l1)))
        (cond ((not (equal n (expt 2 m))) (error "Bad size.")))
        (setq j 1 i 1)
13      (cond ((< i j)
               (setq tr (svref ar j) ti (svref ai j))
               (setf (svref = j) (svref = i))
               (setf (svref ai j) (svref ai i))
               (setf (svref = i) tr)
               (setf (svref ai i) ti)))
        (setq k nv2)
16      (cond ((< k j)
               (setq j (- j k) k (/ k 2))
               (go 16)))
        (setq j (+ j k)
              i (+ i 1))
        (cond ((< i n)
               (go 13)))
        (do ((l 1 (+ l 1)))
            ((> l m))
            (setq le (expt 2 l)
                  le1 (floor le 2)
                  ur 1.0
                  ui 0.0
                  wc (cos (/ 3.14159265 le1))
                  wi (sin (/ 3.14159265 le1)))
            (do ((j 1 (+ j 1)))
                ((> j le1))
                (do ((i j (+ i le)))
                    ((> i n))
                    (setq ip (+ i le1)
```

```
                    tr (- (* (svref ar ip) ur)
                          (* (svref ai ip) ui))
                    ti (+ (* (evref ar ip) ui)
                          (* (svref ai ip) ur)))
                (setf (svref ar ip) (- (svref ar i) tr))
                (setf (svref ai ip) (- (svref ai i) ti))
                (setf (svref ar i) (+ (svref ar i) tr))
                (setf (svref ai i) (+ (svref ai i) ti))))
          (setq tr (- (* ur wr) (* ui wi))
                ti (+ (* ur wi) (* ui wr))
                ur tr ui ti) )
       (return t)))

(defun fft-bench () (dotimes (i 10) (fft re im)))
```

VAX8650/Real Number Lisp	3.6 (See)	
VAX760/Real Number Lisp	19.3	
VAX750/Real Number Lisp	32.5	

Data from Reference [4] , and others

VAX8650/Real Number Lisp	3.6 (See)	(Double precision)
Symbolics3600+IFU	3.87	(Single precision)
DORADO	1.57(CPU) + 3.0 (GC)	
SAIL	4.0 (CPU) + 2.9 (GC)	
PSL-IBM3081	7.30(CPU) + 4.0 (GC)	
VAX780/Real Number Lisp	19.3	
InterLisp/VAX780	22.79	
SUN3/160 Lucid Lisp	34.96	
PSL-DEC20	35 .44(CPU) + 10.8 (GC)	
IBM PC/RT Lucid Lisp	48.67	
Data General MV10000 CL	62.78	
VAX780/CL	32.7 (CPU) + 35.6 (GC) (*2)	
PSL-VAX780	60 .55(CPU) + 13.4 (GC)	

note. (*1) The variable nm1 is set but not used, as is original program.
note. (*2) This measurement was made under different conditions.
It used an array of floating point number, which improves
the code for array indexing and numeric operation.
See reference 2 for the measurement condition.

Figure 7. Continued.

```
;;; Lisp test program.
(defun dif (u)
  (let ((time 0.0) (x 1.0) (vx 0.0))
     (loop (when (>= time 1.0) (return x))
           (setq
                 time (+ time u)
                 x(+ x(* u vx))
                 vx (+ vx (* u -9.8))
           ))))

(defun dif -test () (dif 1. OE-5) )
```

```
c Fortran test program.
        program dif
        double precision time, x, vx, u
        cell lib$init_timer
        call lib$stat_timer[2, itime)
        u = 1.OE-5
        time = 0.0
        x = 1.0
        vx = 0.0
 100    if (time .ge. 1.0) goto 200
        time = time + u
        x = x + u * v x
        vx=vx+u   *-9.8
        goto 100
 200    call lib$stat_timer(2, iitime)
        print *, ' cpu= ', (iitime-itime)*10, ' msec'
        end
```

VAX8650/Fortran	1.4 (See)
VAX8650/Real Number Lisp	1.9
IBM PC/RT C	3.3
VAX780/Fortran	6.0
SUN3/160 C	6.3
VAX780/Real Number Lisp	7.5
VAX750/Fortran	9.5
VAX750/ResJ Number Lisp	13.9
SUN3/160 Lucid Common Lisp	34.98
IBM PC/RT Lucid Common Lisp	57.20

Figure 8. Differential equation program and measurement

NO INTEGER NUMBERS

Our implementation performs floating-point calculation even when integer calculation is adequate. Even array-indexing and simple counting is done with floating-point numbers, though this introduces some inefficiency.

On the VAX8650, there is still a significant difference in speed between integers and double-precision numbers. Our implementation cannot surpass VAX Fortran for this reason.

Advancing VLSI technology is making floating-point operations faster and faster. For example, on a Cray computer a floating-point operation is faster than a memory reference or a branch. Floating-point-number addressing may be feasible on such machines. Reference 4 contains more information about relevant aspects of Cray characteristics.

ARRAY INDEXING

It is true that arrays and floating-point calculation go together. The example of image processing quoted earlier is one instance of this. In spite of this fact, Real Number Lisp still has a weak spot in connection with array indexing.

This problem could be solved if this system provided a vector mapping mechanism similar to that of APL, which has more expressive mapping operators than Common LISP. In APL, most vector operations have no indexing variable; indexing was hidden by the implementer. In that case, the implementer could safely eliminate both the range-checking and number-format conversion on array indexing.

REPLACING BRANCH OPERATIONS WITH EXCEPTIONS

Since exceptions are used for type-checking, the number of branch instructions is reduced. The generated code looks like Fortran code for numeric functions. This technique makes compiled code compact and efficient.

Though exceptions are not quite equivalent to the branches they replace, they are usable for type-checking. VAX computers are able to prevent further instruction executions after exceptions from the arithmetic unit. This is not always the case for other pipelined computers. However, an exception is signalled before the result is used. Therefore, invalid operands are detected before the results are stored in the memory if the compiler generates calculation instructions and store instructions separately. Even if many operations are done concurrently, it is possible to detect type errors before the final result is moved into memory.

IEEE FLOATING FORMAT

This technique is compatible with IEEE floating-number format. Clearly floating-point formats that used all possible bit patterns as valid numbers would not work. However, IEEE defines NaN (Not a Number) as a situation where exponent bits are all set and at least one of the fraction bits is set. If the most significant bit is set, there are still 50 bits remaining, which are enough for non-numeric data.[5]

Some processors such as the Intel 8087, 80287, 80387, 80486 or MIPS Computer System Int's R3010 [6] signal exceptions on NaN operations. This technique is directly applicable for these processors. Even for other processors which just set status flags

on NaN operation, the cost of type-checking is reduced, as this is done by the floating-point processors. There is thus no need to transfer the data from co-processor to main processors for type-checking.

LINKING TO OTHER LANGUAGES

According to the experience with an actual application, [7] Real Number Lisp has good characteristics for linking to external routines. All vectors are compatible with other languages such as Fortran, and external routines written in Fortran are able to detect invalid operation on non-numeric data. These make the whole system robust and safe.

CONCLUSION

It is possible to implement a floating-point-number-oriented LISP on general-purpose computers. Putting every number into double-precision floating format and replacing type-checking codes with exceptions are feasible on large computers which have fast floating arithmetic circuits. Though several problems exist for this implementation such as memory usage, array indexing and language compatibilities, the technique is applicable to systems where both numeric and symbolic computations are required at the same time.

ACKNOWLEDGEMENT

The author thanks the referees of this journal for the valuable advice and help that have been offered in the reviewing process. The measurement data using IBM PC/RT and SUN3/160 have been provided by one of the referees. The author also thanks Ikuo Takeuchi (Nippon Telegraph and Telephone, Basic Research Laboratories) for his suggestions and guidance about implementation techniques.

REFERENCES

1. Digital Equipment Corporation, *VAX Architecture Handbook*, Digital Press, 1977.
2. Richard P. Gabriel, *Performance and Evaluation of Lisp Systems*, MIT Press, Cambridge, Massachusetts, 1985.
3. Guy L. Steele Jr., *Common Lisp, the Language*, Digital Press, 1984.
4. Cray Research, Int, *CAL Assembler Reference Manual, 1985.*
5. ANSI/IEEE, 'IEEE standard for binary floating-point arithmetic', *ANSI/IEEE Standard 754-1985.*
6. Gerry Kane, *mips RISC ARCHITECTURE*, Prentice Hall, 1988.
7. Takeshi Shakunaga, '3-D scene modeling based on a generic model', *5th International Conference on Image Analysis and Processing*, Positano, Italy, 1989.