Stack Techniques and Suppression of A-list for LISP Interpreter

Chiaki HISHINUMA*, Kenji YAMASHITA** and Masakazu NAKANISHI***

ABSTRACT—Some useful techniques for implementing a LISP interpreter on minicomputers are described. First, effective stack techniques are proposed to realize recursive procedures peculiar to the functions of LISP. Second, a method is proposed to suppress the excess a-list which is generated by universal functions when they evaluate EXPR functions of iterative form.

1. INTRODUCTION

LISP language has been widely used to develop non-numerical algorithms or to solve non-numerical problems. Especially, the implementation of LISP on a small computer is very useful to develop some basic algorithms or to offer convenient tools for educationl use of LISP language and recursive programming techniques.

LISP, however, has several problems of nonefficiency in processing: The rapid increasing of the stack area caused by recursive calling of procedures and the relatively low speed of processing are the typical examples of the problems. Moreover, the data structure formed by binary lists can not contain so much information in a unit memory space and the universal functions waste many list elements only for their operation.

In this paper, we propose some useful techniques for implementing LISP interpreters which improve the processing speed and realize an efficient use of list structure, without changing the language specifications of LISP 1.5 interpreter, by improving the processing algorithm of the interpreter.

2. STACK TECHNIQUES

The efficiency of LISP interpreter much depends on how to realize recursive procedures using a stack. The following programming techniques are effective for the realization of the built-in functions of LISP interpreter.

1) Return address and arguments are pushed onto the stack by the called procedure and are stored in this order.

This paper first appeared in Japanese in Joho－Shori (Journal of the Information Processing Society of Japan), Vol. 17, No. 11 (1976), pp. 1002～1008.

* Musashino Electrical Communication Laboratory Nippon Telegraph and Telephone Corporation
** Bridgestone Tire Company Ltd.
*** Faculty of Engineering, Keio University

2) For iterative calling procedures* of SUBR and FSUBR functions, it is not necessary to stack new parameters, i.e. return address and arguments. Therefore, the parameters are changed in the stack directly, if necessary.

3) The return address which can be known by the called procedure are not pushed onto the stack.

4) Parameters, which are already referred to and will be never referred to again, are deleted from the stack.

By extensively using the above techniques, an interpreter is realized where only small size is required for the stack, faster stack operation is accomplished and used lists are set free in an early stage of computation.

3. SUPPRESSION OF A-LIST

Let us take an example of the following EXPR function definition.

$$reverse[x]=rev1[x;NIL]$$
$$rev1[x;u]=[null[x]\rightarrow u;$$
$$t\rightarrow rev1[cdr[x];cons[car[x];u]]]. \qquad (1)$$

The above definition can also be written as follows using program feature.

$$reverse[x]=prog[[u];$$
$$L\ [null[x]\rightarrow return[u]];$$
$$u:=cons[car[x];u];$$
$$x:=cdr[x];$$
$$go[L]$$
$$]. \qquad (2)$$

By the definition of the universal function of LISP 1.5, the program (1) might cause a fatal situation, i.e. storage overflow, if x is an awfully long list. That is, in evaluating $reverse[x]$ defined by (1), the maximum necessary length of a-list is $2|x|+1$, where $|x|$ denotes the length of list x. But in evaluating $reverse[x]$ defined by (2), the maximum length of a-list is 2 for any list x. Therefore, it can be said that the program (2) is more desirable than (1) for a small LISP system because of its small size requirement on free list area. But the program using program feature might lose the elegance and easy-to-program capability of LISP language.

We shall discuss a method in the following sections for implementing an inter-

---

* In general, if

$$f(x_1,\ldots,x_n)=\varepsilon(f,x_1,\ldots,x_n,g_1,\ldots,g_m)$$

where ε is an expression (usually conditional) and $g_1,\ldots,g_m$ are functions, then f is said to be *iterative* if f never occurs as an argument of one of the g_i. For the detailed definition of iterative procedures, refer to McCarthy.[4]

preter which works as (2), even if *reverse*[*x*] is given in the form of (1).

3.1 IMPROVEMENTS OF THE UNIVERSAL FUNCTION

When an EXPR function is defined iteratively, the increase of a-list can be suppressed by the following method:

When a new pair of variable and its value is generated in applying LAMBDA expression, a check is made whether or not the new pair needs to be really appended to a-list, by investigating the contents of a-list ranging from the pointer in the top of the stack to the pointer in the next-to-top.

Case 1) If the same variable has been found in that range, replace the content of the value part of that pair by the new value without appending the new pair to a-list.

Case 2) If the same variable is not found, append the new pair to a-list (Fig.1).

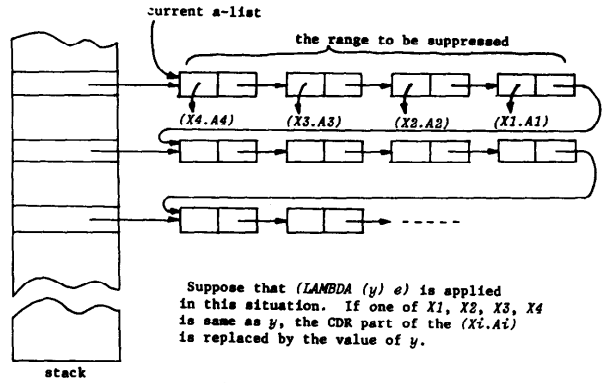

Fig.1 A-list and the stack

The above method corresponds to rewriting the evaluation part of λ-expression in the definition of *apply* of LISP 1.5 as follows:

$$eq[car[fn];LAMBDA]\rightarrow eval[caddr[fn];matchpair[cadr[fn];args;a]]$$

where *matchpair* is defined as follows:

$$\begin{aligned} &matchpair[u;v;a]\\ &\quad=[null[u]\rightarrow[null[v]\rightarrow a;t\rightarrow error[F2]];\\ &\qquad null[v]\rightarrow error[F3];\\ &\qquad ask[car[u];car[v];a]\rightarrow matchpair[cdr[u];cdr[v];a];\\ &\qquad t\rightarrow cons[cons[car[u];car[v]];matchpair[cdr[u];cdr[v];a]] \end{aligned} \tag{3}$$

$$\begin{aligned} &ask[x;y;a]\\ &\quad=[null[a]\rightarrow NIL;\\ &\qquad eq[a;\$A]\rightarrow NIL;\\ &\qquad eq[caar[a];x]\rightarrow rplacd[car[a];y];\\ &\qquad t\rightarrow ask[x;y;cdr[a]]], \end{aligned} \tag{4}$$

where $\$A$ is the pointer to a-list of the last level in the stack.

3.2 SIDE EFFECTS AND THEIR SETTLEMENTS

The method explained above may cause some side effects.

First, let us investigate the following example:

$$g=\lambda[[x;y];h[y;function[\lambda[[y];cons[x;y]]]]],$$

$$h = \lambda[[x;fn];fn[x]]. \qquad (5)$$

In the definition of the function g, the function h is referred to iteratively and the functional argument given to h has the free variable x which is the same name as the formal argument of the definition of h.

The value of $g[A;B]$, for instance, should be $(A.B)$ by definition. However, the method explained above makes it $(B.B)$ by the operation of a-list suppression in evaluating the function *FUNARG*. This type of side effect can be settled by copying the portion of a-list of $(FUNARG\ fn\ a)$ between the last level and the current level when $(FUNCTION\ fn)$ is evaluated. It can be expressed by the following modification of a part of *eval*.

$$eq[car[form];FUNCTION]\rightarrow list[FUNARG;cadr[form];copya[a;\$A]],$$

where *copya* is defined by

$$copya[x;y]=[eq[x;y]\rightarrow x;\ t\rightarrow cons[car[x];copya[car[x];copya[cdr[x];y]]]]. \qquad (6)$$

Another side effect may appear in the case of EXPR function of the following type:

$$\begin{aligned} &s=\lambda[[x;y];g[y;function[\lambda[[y];h[x;y]]]]],\\ &g=\lambda[[y;fn];cons[fn[car[y]];fn[cdr[y]]]],\\ &h=\lambda[[y;x];cons[y;x]], \end{aligned} \qquad (7)$$

where the functional argument refers to h iteratively with the free variable x which is the same name as the formal argument used in the definition of h.

The value of $s[A;(1\ 2)]$ should be $((A.1)(A.2))$ by definition, but by the method discussed in 3.1 it will be $((A.1)(1.2))$ as a result of suppressing a-list of $(FUNARG\ fn\ a)$ which is created in evaluating $(FUNCTION\ fn)$. This side effect can be settled by changing the part which processes *FUNARG* in *apply* as follows:

$$eq[car[fn];FUNARG]\rightarrow i[apply[cadr[fn];args;cadr[fn]]],$$

where i is an identity function which generates a new element of the stack, that is, when an argument of i is evaluated, the level of stack is raised to higher than the current.

4. PERFORMANCE OF THE METHODS

Table 1 shows the comparative data of the execution time and the number of garbage collections for some test programs executed by the mini-LISP, which is made to employ the methods discussed in this paper, and by the other three LISP 1.5 interpreters. These test programs were used at the LISP contest (Symposium on symbolic

manipulation, IPSJ, 1974). The mini-LISP has excellent performance not inferior to other large LISP systems in spite of its small memory capacity.

Table 1. Comparison of the execution time of LISP 1.5 interpreters

System name		mini-LISP a-list suppressed	mini-LISP non-suppressed	KLISP	MLISP	OLISP
machine		HITAC10 (4kw)		TOSBAC 3400 30	MELCOM 7700	NEAC 2200
stack(words) / free cells		α / 830-2α		744 / 3,936	2,500 / 25,000	29,000
Test programs msec (no. of GC)	WANG A	71(0)	66(0)	40(0)	330(0)	43(0)
	WANG B	402(0)	433(1)	260(0)	630(0)	123(0)
	Bit A*	1,175(1)	1,170(2)	560(0)		232(0)
	Bit B*	1,211(1)	1,260(2)	380(0)	1,850(0)	177(0)
	Sort	58,044(135)		28,940(11)	62,333(3)	35,457(9)
REMARKS		*mapcar, mapcon are EXPR			*mapcar is EXPR	

5. CONCLUSION

Some useful methods for implementing LISP 1.5 interpreter are described. First, effective stack techniques to prevent the rapid increasing of the stack area caused by recursive calling of procedures are proposed. Second, the fact is indicated that the excess a-list may be generated in the evaluation of EXPR functions of iterative form, and a method is proposed for suppressing such a-list. The performance of the methods proposed here was also shown to be satisfactory by comparing the mini-LISP which is made to employ our methods on a minicomputer with other LISP 1.5 interpreters.

ACKNOWLEDGEMENT

The authors wish to thank Prof. S. Mori of Keio University for his hospitality and encouragement.

REFERENCES

(1) McCarthy,J., et al.: LISP 1.5 Programmer's Manual, The MIT Press (1962)

(2) Nakanishi,M.: LISP contest, bit, vol.7, no.3 (1975)

(3) Nakanishi,M., Hishinuma,C., Yamashita,K., and Sakai,T.: A design of LISP interpreter for minicomputers, Minicomputer Software, North-Holland (1976)

(4) McCarthy,J.: Towards a mathematical science of computation. In 'Information processing 1962', Proceedings of the IFIP Congress, 1962. North-Holland (1963)