Short Note

A Primitive for Non-recursive List Processing

Masayuki Suzuki,* Kiyoshi Ono* and Eiichi Goto*,**

A new LISP primitive *rcons* (*r*everse *cons*) is proposed, which can be used to transform a certain type of recursively defined functions into iterative ones.

Rcons, which constructs a list in the reverse order (i.e., from head to tail), could be defined in terms of *rplacd* and *cons*.

However, as a new primitive instead of a composite function, *rcons* can dispose of overheads in space and time due to *rplacd*, used inside the composite *rcons*, in *cdr*-coding implementations.

1. Introduction

Two-bit *cdr*-coding LISP system was proposed [2, 5] so as to represent lists compactly: a list is usually represented as consecutive cells (a *linear list*), containing its elements (*car* parts), without *cdr* parts, which would connect the elements in conventional LISP systems.

A primitive constructor *cons* in the system can construct *linear lists* in the direction from tail to head. Although *cons* seems sufficient for recursively defined functions, improved iterative versions of the functions sometimes require a list to be constructed in the reverse direction (see Section 2). However, *cons* or a combination of existing primitives cannot construct *linear lists* in the reverse direction.

This note proposes a new primitive constructor *rcons* (*r*everse *cons*), which constructs *linear lists* in the direction from head to tail. We think that *rcons* has two significances:

1) *Rcons* has an intuitive meaning as another constructor complementing *cons* especially in recursion elimination (Section 2).

2) *Rcons* as a primitive can be implemented in *cdr*-coding systems as efficiently as *cons* (Section 3).

2. *Rcons* as Another Constructor and its Application to Recursion Elimination

We could define *rcons* as a composite function, which has the same semantics as a more efficient implementation described in Section 3.

```
rcons [x; y]=
        [null[x]→cons[y; NIL];
         atom[x]→error[  ];
         T→cdr[rplacd[x; cons[y; NIL]]  ]  ]
```

Note that *rcons* appends a new element at the end of an existing list by redirecting the *cdr* part of the last element of the existing list. Compare *rcons* with *cons* in Fig. 1.

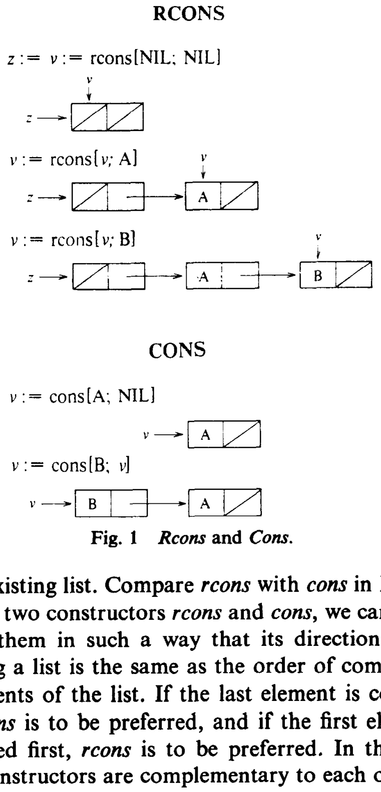


Fig. 1 *Rcons* and *Cons*.

From two constructors *rcons* and *cons*, we can choose one of them in such a way that its direction of constructing a list is the same as the order of computation of elements of the list. If the last element is computed first, *cons* is to be preferred, and if the first element is computed first, *rcons* is to be preferred. In this sense, these constructors are complementary to each other.

Rcons is applicable to eliminating recursive functions of a certain pattern, which is one of those described by Risch [8]:

```
f[x; y]=[null [x]→s[y];
            p[x; y]→f[cdr[x]; y];
            q[x; y]→cons[g[x; y]; f[cdr[x]; y]]]
```

This pattern appears in many list handling recursive LISP functions, such as *append*, *union*, and *intersection* [7].

The pattern can be transformed with *rcons* into:

```
f[x; y]=prog[[v; z]
    z: =v: =rcons[NIL; NIL];
```

*Department of Information Science, Faculty of Science, University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo 113, Japan.
**The Institute of Physical and Chemical Research, 2-1, Hirosawa, Wako-shi, Saitama 351, Japan.

```
Loop
[null[x]→return[prog2[rplacd[v; s[y]]; cdr[z]]];
  p[x; y]→NIL;
  q[x; y]→v := rcons[v; g[x; y]]];
x := cdr[x]; go[Loop]]
```

3. *Rcons* in Cdr-coding Systems

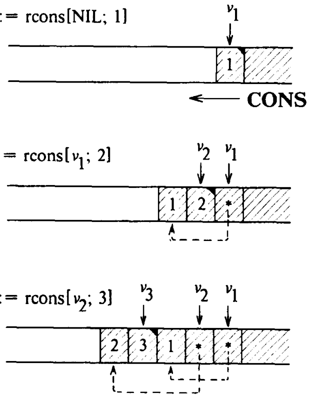
In *cdr*-coding systems, *rcons* defined in the previous section might be time and space consuming owing to *rplacd*, which usually requires an introduction of *invisible cells*, and hence impairs the advantages of *cdr*-coding systems, which intend to eliminate the space for *cdr* parts. In other words, *rcons* thus defined could not exploit the full advantages of both the recursion elimination with *rcons* and *cdr*-coding systems. (See Fig. 2.)

However, *rcons* can be made as efficient as *cons* if it is turned into a new primitive, taking advantage of a storage allocation mechanism of *cdr*-coding systems.

In *cdr*-coding systems, a cell is usually allocated by *cons* from one end of a *free storage* so that as many elements of a list as possible reside side by side. On the contrary, *rcons*, as a new primitive, can be arranged to allocate a cell from the opposite end of the *free storage* so that the order of allocation be the same as the order of elements in a list.

In Fig. 3, *cons* allocates cells from the right end of the free storage to the left, whereas *rcons* allocates cells from the left to the right. The area containing cells allocated by *rcons* will be called R-area. Note that lists in R-area have the same structure as those in usual free storage area, and where the lists reside is transparent to LISP users.

Conventional LISP primitives, such as *car*, *cdr* and *cons*, and garbage collector can be implemented by the method, with little modification, described by Hansen.



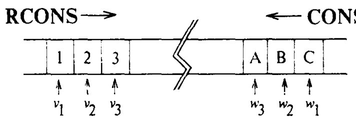
1) v_1 := rcons[NIL; 1]
v_1
1
CONS
2) v_2 := rcons[v_1; 2]
v_2 v_1
1 2
3) v_3 := rcons[v_2; 3]
v_3 v_2 v_1
2 3 1

* invisible cell

◥ end of a list terminated with *NIL*

▨ used cells

Fig. 2 *Rcons* composed of *cons* and *rplacd*.



1) v_1 := rcons[NIL; 1]	w_1 := cons[C; NIL]
2) v_2 := rcons[v_1; 2]	w_2 := cons[B; w_1]
3) v_3 := rcons[v_2; 3]	w_3 := cons[A; w_2]

values of variables after step 3

v_1 = (1 2 3)	w_1 = (C)
v_2 = (2 3)	w_2 = (B C)
v_3 = (3)	w_3 = (A B C)

Fig. 3 Growing direction of linear lists by *cons* and *rcons*.

Especially, after garbage collection, which is invoked when the front ends of R-area and the usual area meet each other, all cells previously in R-area are moved to the usual area.

Fig. 4 shows an example of *rcons*[x; y], where there are three cases depending on where the value of x points to:

1) The value of x points to the front end of R-area.
2) The value of x points to the inside of R-area.
3) The value of x points to the usual area.

Although cases 2 and 3 require an introduction of an invisible cell, the subsequent call on *rcons* will be case 1 and the invisible cell may be avoided.

4. Concluding Remarks

A new LISP primitive *rcons* is proposed, which can be used to transform a certain pattern of recursively defined functions into iterative ones.

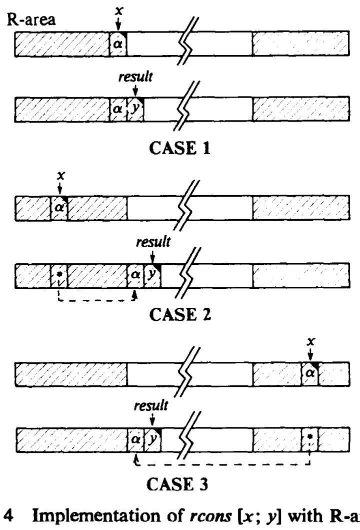


Fig. 4 Implementation of *rcons* [x; y] with R-area.

Although *rcons* could be defined in terms of *rplacd* and *cons*, it is advantageous in *cdr*-coding systems to introduce *rcons* as a new primitive, which combines the effects of *cons* and *rplacd* in such a way that no overhead is incurred in space and time due to *rplacd*.

Our idea is to arrange for a primitive *rcons* to allocate a cell in R-area, which grows in the direction from one end of a *free storage* to the other from which *cons* starts to allocate cells. Namely, R-area allows lists to be linearly constructed from head to tail. (Without R-area, lists could not be linearly constructed from head to tail in a usual *cdr*-coding system. At garbage collection time, however, lists could be rearranged with time-consuming linearization phase.)

Since most previous studies on recursion eliminations have been concentrated on elimination of recursively defined procedures [1], and little effort has been devoted to systematically eliminate recursively defined functions, whose resulting value is a data structure, such as linked lists [3, 8], it is not yet clear whether other new primitives will have more general applications in recursion eliminations. However, several papers have appeared on individual problems, such as non-recursive list copying operations [6]. The individual problems were tackled, case by case, with ingenuity including the use of data structure itself as a stack by modifying the *car* and *cdr* parts of the existing data structure appropriately. It will be interesting future studies to develop recursion elimination techniques and to invent, at the same time, new primitives which conform well to existing systems or which suggest a new system architecture [4].

References

1. Bird, R. S. Notes on Recursion Elimination. *Comm. ACM* **20**, 6 (June 1977), 434–439.
2. Bobrow, D. G. and Clark, D. W. Compact Encodings of List Structure. *ACM Trans. Programming Languages and Systems* **1**, 2 (Oct. 1979), 266–286.
3. Darlington, J. and Burstall, R. M. A System which Automatically Improves Programs. *Acta Informatica* **6**, (1976), 41–60.
4. Goto, E., Ida, T., Hiraki, K., Suzuki, M. and Inada, N. FLATS, A Machine for Numerical, Symbolic and Associative Computing. *Proc. of the 6th Annual Symposium on Computer Architecture*, April 23-25, (1979), 102–110.
5. Hansen, W. L. Compact List Representation: Definition, Garbage Collection, and System Implementation. *Comm. ACM* **12**, 9 (Sept. 1969), 499–507.
6. Lee, K. P. A Linear Algorithm for Copying Binary Trees Using Bounded Workspace. *Comm. ACM* **23**, 3 (March 1980), 159–162.
7. McCarthy, J., et al. *LISP 1.5 Programmer's Manual*. M.I.T. Press, Cambridge, Mass., (1962).
8. Risch, T., REMREC—A Program for Automatic Recursion Removal in LISP. Datalog. Report No. DLU 73/24, Uppsala, Sweden (1973).

(Received April 3, 1981: revised July 19, 1981)