DATALOGILABORATORIET
Inst för informationsbehandling
Uppsala universitet

1973-11-13

DLU 73/24

REMREC - A program for automatic recursion removal in LISP

by Tore Risch

1. Abstracts

It is well known that recursive functions are very common in LISP. Recursive code is mostly the easiest to write and to analyse. However, situations often arise when it is preferable to let the computer work with non-recursive functions. Examples of this are in compiled code and on stack overflow. It is then very often possible to remove recursion (at least partly) in the functions, without therefore introducing stacks in the new code.

Remrec is a program, which automatically transfers some classes of recursive LISP-functions into equivalent non-recursive ones, without introducing stacks.

2. Previous work

Earlier practical programs for recursive removal are the BBN-LISP-compiler (4) and "A system which automatically improves programs" by Darlington and Burstall (2). Strong (3) has studied recursion removal theoretically, but no implementations have been described.

3. REMREC

3.1 Recursion types

The type of recursion which REMREC can remove are divided into 7 classes, according to some simple schemes like

```
f(x) = if p(x) then r(x)
       else h(... f(g(x))...)
```

where the choice of h and the argument position in h where f appears decides the recursion type. The classes are:

R. Primitive recursion.
Scheme:

```
f(x) = if p(x) then r(x)
       else f(g(x))
```

RCONS: REcursion under the last argument in functions which build up list structures. (cons,append,list etc)
Scheme:

```
f(x) = if p(x) then r(x)
       else h(q(x), f(g(x)))
```

where h is cons, append and others

RTIMES,RPLUS: Recursion under times and plus.
Scheme:
Like type RCONS, but h is plus or times, although f may appear in any argument position in h.

RAPPEND,RNCONC: Recursion under first argument to append and nconc.
Scheme:

```
f(x) = if p(x) then r(x)
       else h(f(g(x)),q(x))
```

where h is append or nconc.

RFOA: Recursion under Functions with One Argument. This case has been treated by Strong (3) page 3.

Scheme:

```
f(x) = if p(x) then r(x)
       else h(f(g(x)))
```

where h is any function or nest of functions each of which has one non-constant argument.

Type R may be combined with any of the other recursion types.

REMREC can also handle recursion in nested conditional expressions, functions with several arguments and some other more complicated cases.

Often one function involves recursion of several classes, where REMREC cannot remove both, but has to choose. REMREC then removes the recursion type that

1. appears the most frequently
2. usually generates the simplest non-recursion code.

In addition the user can direct REMREC to remove exactly the recursion type he wants.

3.2 Methods used

REMREC works in two steps. During step 1 REMREC builds up an AND/OR-tree which describes how and where the removable recursion appears in the code. This "recursion tree" has the following properties:

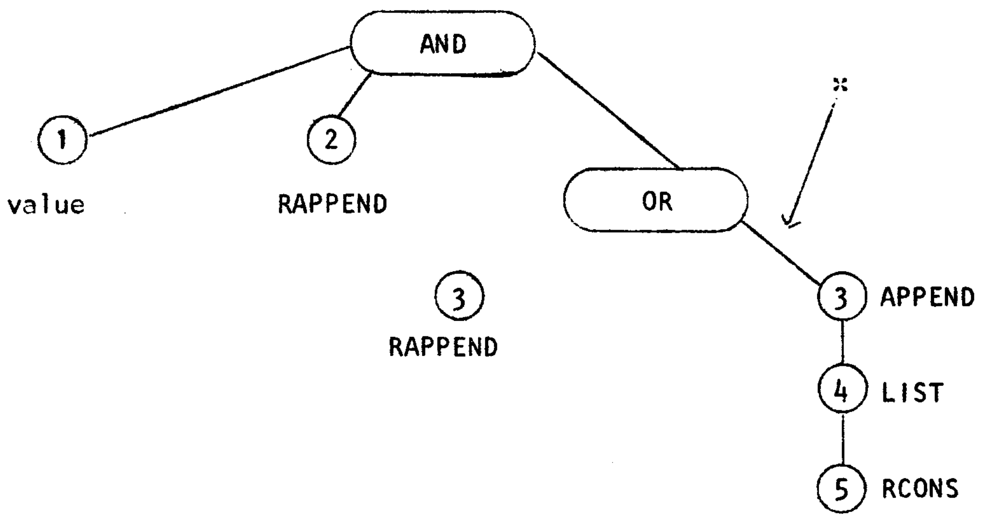
1. The leaves are:
   (a) Pointers to place in the code where "legal" recursion appears, associated with markers for the recursion type.
   (b) Pointers to non-recursive branches in such conditional expressions, where some other branch is recursive.
2. When OR-nodes appear it is possible to remove recursion in several ways.
3. The AND-nodes describe conditional expressions in a broad sense. (_cond_, _and_, _or_ and _selectq_-expressions)

4. There are also pointers with markers to other interesting places in the code "on the way" to a recursive call.

   Example of a "recursion tree". Consider the LISP-function:

```
totreverse(x) == if null(x) then NIL ——— (1)
                 elseif atom (car(x))            (2)
                        then append(totreverse(cdr(x)),
                        list(car(x)))
                 else append (totreverse(cdr(x)),
                        (3)  list(totreverse(car(x))))
                           (4)       (5)
```

   It has the "recursion tree":



   (REMREC can remove recursion under "chains" of function calls in the last argument to append, list, cons, nconc, rplaca and rplacd. In the example we have a "chain" of append and list)

A counter is associated with each recursion type. In the example above the counter RCONS = 1 and RAPPEND = 2. During step 1 REMREC also checks so that side effects will not prevent the recursion removal.

During step II REMREC

1. Looks for the counter with the highest value, or (in case of a tie) the one which appears first on a priority list. That counter decides which recursion type shall be removed.

2. Cuts off those branches in the tree which are no longer of interest. (E.g. * in the example above). In the simplest case this means that the unused branches in OR-nodes are cut off. After this the tree contains pointers only to the places in the code where the code shall be changed.
3. Searches through the tree (preorder) and changes the code at all nodes.

3.3 Use of REMREC

REMREC is constructed with the goal to be user friendly and that the generated code shall be as efficient as possible. The appearance of side effects can in many cases influence the recursion removal. REMREC therefore works in 4 modes, where the side effects are treated differently. Of these modes one is interactive, one is intended to be used at calls from other programs and two are 'batch'-modes. In interactive mode REMREC consults the user about side effects at sensitive points. The user can declare the functions with side effects. Important is that the code, produced by REMREC will in all cases give the same result as the original one.

3.4 Examples of runs

Below there are examples of functions involving different recursion types, to which REMREC has been applied. REMREC operates on the internal LISP-representation for programs (S-expressions), but for readability reasons the examples are expressed in an Algol-like notation.

Recursion type R:

Original code:
```
even(1) == if null(1) then T
           elseif null(cdr(1)) then NIL
           else even(cddr(1))
```

After applying REMREC:
```
even(1) == proc ( NIL
     even: if null(1) then return(T)
           elseif null(cdr(1)) then return(NIL)
           else begin 1:=cddr(1);
                      goto even;
                end )
```

Recursion type RTIMES

Original code:

```
   fak(n) == if n=0 then 1
             else fak(n-1) * n;
```

After applying REMREC:

```
   fak(n) == prog( (e0)
             e0:=1;
        fak: if n=0 then return(e0)
             else begin e0:=n*e0;
                        n:=n-1;
                        goto fak;
                  end )
```

Recursion type RPLUS:

Original code:

```
   scount(x,1) == if atom(1) then 0
                  elseif x = car(1) then addl(scount(x,cdr(1)))
                  else scount(x,car(1)) + scount(x,cdr(1));
```

After applying REMREC:

```
   scount(x,1) == prog ( (E0)
                  E0:=0;
          scount: if atom(1) then return(E0)
                  elseif x=car(1) then
                      begin E0:=addl(E0); 1:=cdr(1); goto scount
                      end
                  else begin E0 := scount(x,car(1)) + E0;
                             1:=cdr(1); goto scount;
                       end );
```

Recursion type RNCONC:

Original code:

```
   reverse(x) == if null(x) then NIL
                 else nconcl(reverse(cdr(1)),car(1))
```

After applying REMREC:

```
   reverse(x) ==prog( (B2)
       reverse: if null (x) then return(B2)
                else begin B2:=cons(car(1),B2);
                           1:=cdr(1);
                           goto reverse;
                     end ; )
```

Recursion type RCONS:

Original code:

```
snitt(x,y) ==
   if null(x) then NIL
   elseif member(car(x),y) then cons(car(x),snitt(cdr(x),y))
   else snitt(cdr(x),y)
```

After applying REMREC:

```
snitt(x,y) ==
   prog( (E0 B1)
      B1:=E0:=cons(NIL,NIL);
   snitt: if null(x) then begin rplacd(E0,NIL);
                                return(cdr(B1));
                          end
   elseif member(car(x),y) then
       begin rplacd(E0,cons(car(x),NIL)
             Eo:=cdr(E0);
              x:=cdr(x);
              goto snitt;
          end;
       else begin x:=cdr(x);
             goto snitt;
            end )
```

4. Comparison with other programs

The BBN-LISP-compiler involves a simple recursion remover. It can remove trivial recursion (type R).

The system made by Burstall and Darlington (2) handles more general (and more difficult) cases than REMREC. It involves among others a theorem prover and a pattern matcher. Each recursion type has:

1. A pattern which is matched against the actual function.
2. Some relations which shall be satisfied.

REMREC has two advantages compared to this system:

1. It works on full LISP, while (2) works on a subset.

2. Although the pattern method is quite general, some of the recursion which REMREC can remove is very difficult to represent with patterns. E.g. when recursion appears under nested function-calls, or in several branches of a conditional expression.

5. Desirable developments of REMREC

It is relatively easy to add code to REMREC so that also other types of recursion than those mentioned above can be removed. Today REMREC removes the recursion types (except recursion under prog) which I think are the most common in LISP and whose iterative code is not too complicated. The following developments are desirable in REMREC:

1. Recursion under prog, and other recursion types ought to be added in REMREC. E.g. the type Strong (1) was given on page 8.
2. The treatment of side effects should be more advanced. Today the user has to tell the systems the functions (sometimes forms) which have side effects. In a later version REMREC could itself find significant side effects in many cases.
3. REMREC ought to be able to remove recursion in systems of functions.
4. REMREC could be a part of a bigger system for function improvment and manipulation which would also contain REDFUN (3) and other function manipulation programs. In such a bigger system some improvments are very easy to add directly in REMREC. E.g., changing of recursion under cons to recursion under rplacd ((2) calls that "run-time garbage collection").

References

1. H.R. Strong
"Flowchartable Recursion Specifications:"
IBM Thomas J. Watson Research Center,
memo No. RC3322 (April 1971)

2. J. Darlington and R.M. Burstall
"A System whichAutomatically Improves Programs"
IJCAI, 1973

3. Erik Sandewall, Anders Haraldson, Arne Tengvald
"Documentation of the REDFUN package"
(July 1971)

4. D.G. Bobrow et al
BBN-LISP TENEX reference manual
Bolt Beranek and Newman Inc.
Cambridge, Mass. (July 1971)

Appendix: Efficiency messurements

I have made a number of tests to compare the execution times for some simple LISP-functions before and after applying REMREC. The tests were made at BBN-Uppsala-LISP in October 1973.

The functions I have tested are a sort of "type" functions for each recursion type. The timings therefore ought to be some of the most advantageous for REMREC.

First there follows a list of the definitions of the tested function before and after applying REMREC. After that the execution times are shown. (The numbers within parenthesis refer to the execution time tables.

Type RCONS (1)

Original code:

```
    copyl(x) == if null(x) then NIL
                else cons(car(x),copyl(cdr(x)));
```

After REMREC:

```
    copyl(x) == prog( (B E)
                B:=E:=cons(NIL,NIL);
         copyl: if null(x) then begin cdr(e):=NIL;
                                      return(cdr(B)); end
                else begin cdr(E):=cons(car(x),NIL);
                           x:=cdr(x);
                           goto copyl; end; end; );
```

Type RPLUS (2)

Original code:

```
    length(x) == if null(x) then 0
                 else addl(length(cdr (x)));
```

After REMREC:

```
    length(x) ==prog ( (sum)
                SUM:=0;
        length: if null(x) then return(sum)
                else begin SUM:=addl(SUM) ; X:=cdr(x);
                           goto length; end; );
```

Type RNCONC (3)

Original code:

```
    rev(x) == if null(x) then NIL
              else nconcl(rev(cdr(x)),car(x))
```

After REMREC:

```
    rev(x) == prog ( (B)
         rev: if null(x) then return(b)
              else begin B:=cons(car(x),B); x:=cdr(x);
                         goto rev; end)
```

Type RNCONC (4)

Original code:

```
    rev(x) == if null(x) then NIL
              else nconc(rev(cdr(x)),cons(car(x)))
```

After REMREC:

```
    rev(x) == prog ( (B)
         rev: if null(x) then return(b)
              else begin b:=nconc(cons(car(x)),L);
                         x:=cdr(x); goto rev; end )
```

Type RFOA (5)

Original code:

```
    length(x) == if null(x) then 0
                 else adda(length(cdr(x)))
```

After REMREC:

```
    length(x) == prog((B E)
                 E:=0;
         length: if null(x) then begin B:=0; goto L2; end
                 else begin E:=addl(e); goto length; end
             L2: if zerop(e) then return(B)
                 else begin B:=adda(B); E:=subl(E); goto L2; end)
```

Definition of adda:

```
    adda(x) == addl(x)
```

Execution timings:

Each function is tested in 4 ways: Before/after applying REMREC and before/after compiling.

At each test the time to execute the function a number of times (mostly 5) in a loop is noted 3 times. The number within parenthesis below show the medium value of the 3 timings. The standard deviation is given within parenthesis. This method is used to obtain an approximation of the timing error.

	UNCOMPILED	COMPILED
(1)		
Before applying RR	2696	290 (7)
After applying RR	2372 (35)	117 (3)
Time saving:	12%	38%
(2)		
Before applying RR	5196 (62)	293 (14)
After applying RR	3527 (132)	96 (4)
Time saving	32%	67%
(3)		
Before applying RR	3098 (18)	650 (12)
After applying RR	1974 (74)	96 (13)
Time saving	36%	85%
(4)		
Before applying RR	3068 (79)	499 (6)
After applying RR	2259 (34)	252 (8)
Time saving	26%	49%
(5)		
Before applying RR	5091 (162)	592 (28)
After applying RR	7297 (126)	434 (15)
Time saving	-43%	26%