Inter-Office Memorandum

To	MPS Group	Date	August 20, 1973
From	J. Morris	Location	Palo Alto
Subject	Pointer Swinging vs. Node Overwriting	Organization	PARC/CSL

XEROX

Pointer manipulation is tricky. A source of irritation is that a programmer occasionally finds himself one step further down a list than he would like to be. Another is having to fiddle at the beginning or end of a structure or treat the empty structure as a special case. The situation can be ameliorated by taking the CPL view of data structures [S,P]. I was exposed to this view several years ago, but only recently came to appreciate it.

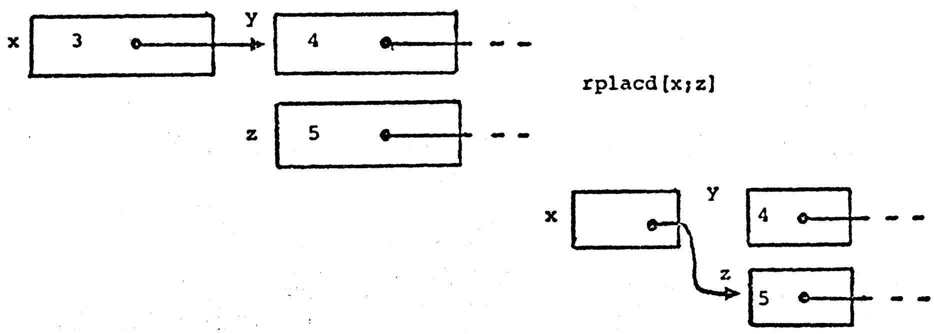
The most common and obvious method of altering a structure is to change the component of a node (e.g. rplacd in LISP) which interpreted graphically amounts to swinging a pointer; i.e. moving its arrowed end. The CPL method is to overwrite the entire node. Graphically this amounts to moving the unarrowed end(s) of one or more pointers at once. Figure 1 illustrates these two kinds of transformation.

It is easy to simulate pointer swinging by node overwriting:

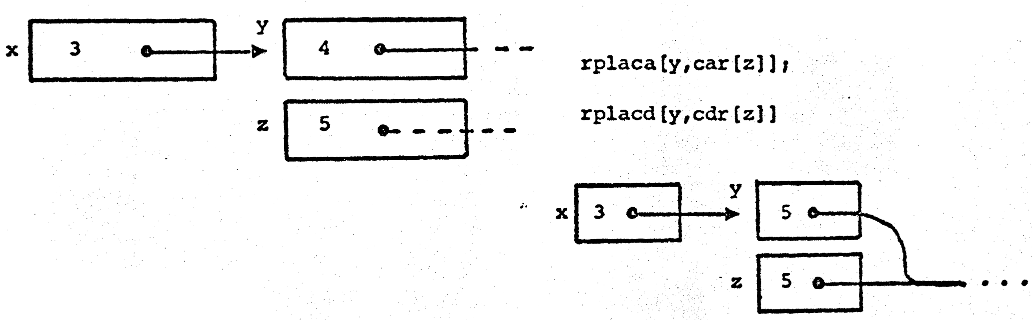
rplacd[x;z] = overwrite x with cons[car[x];z]

It is not so easy to reverse the simulation because the overwrite scheme allows one to change the amount of information in a node. By implication, node overwriting is more expensive to implement, either in terms of space-time or complexity.

Figure 1. Two kinds of structure change.



(a) Pointer Swing



(b) Node Overwrite

In principle node overwriting is supported by any language with union types and reference variables or their equivalents; e.g. ALGOL-68, PASCAL [vW,W]. I shall use PASCAL to illustrate it.

Suppose one wishes to implement lists of integers. He makes the declaration

$$\underline{\text{type}}\ \text{list} = \uparrow\ \underline{\text{record}}\ \text{hd: integer; tl:list}\ \underline{\text{end}}$$

which says that a value of type list is a pointer to a record consisting of an integer and a list. The constant nil is implicitly a pointer of any type and is used to represent the empty list.

If x is declared a list, by

$$\underline{\text{var}}\ \ \text{x:list}$$

the value of

$$x\uparrow$$

is its contents, a record, and the values of

$$x\uparrow.hd \text{ and } x\uparrow.tl$$

are the respective components of the record. Thus getting the tail of a list is a two step process: taking the contents of a pointer and selecting a component of the contents.

There are basically two kinds of assignment

$$x:=y$$

changes the value of x,

$$x\uparrow:=z$$

changes the contents of the pointer x. An assignment like

$$E.hd :=3$$

should be regarded as an abbreviation for

$$E := \langle 3, E.tl\rangle$$

whatever E happens to be. E.g.

$$x\uparrow.hd :=3$$

changes the contents of the pointer x and happens to leave its tl unchanged.

The representation chosen here for lists uses the pointer swinging strategy. It induces the irritations discussed at the beginning, as the following example illustrates.

Suppose one wishes to delete all the odd numbers from a list l. In this representation a deletion must be accomplished by changing the tl of the preceding element. Thus one must hang on to the element preceding the one whose hd he is examining. To make matters worse, if the element is the first one on the list, the deletion must be done by a simple assignment to l. These facts contribute to the opacity of the program:

```
L: if l=nil then goto End;
   if¬odd(l↑.hd) then goto M;
   l :=l↑.tl; goto L;
M: x:=l;
   while x↑.tl≠nil do
       if odd(x↑.tl↑.hd)
          then    x↑.tl := x↑.tl↑.tl
          else    x:=x↑.tl
End:
```

The reader is invited to simplify the program; his taste may suggest using two variables to scan the list, using LISP or ALGOL-W notation to avoid all the "↑."'s, or eliminating the goto's. It's still pretty bad. (A referee who rewrote it to eliminate goto's introduced a bug.)

The cure for the problems is to adopt an "unobvious" representation for lists using the node overwrite strategy.

Statically the change seems quite minor: a list becomes a pointer to a union type half of which is an empty indicator. PASCAL's way of saying this is

```
type list=↑record case empty:Boolean of
                       true :  ;
                       false: (hd:integer;tl:list)
          end
```

The value of

```
x↑.empty
```

will tell one if x is empty.

The dynamics of the situation are quite different. To change a structure one usually overwrites the entire contents of a pointer; e.g.,

```
x↑  := x↑.tl↑
```

removes x↑.hd from a list by changing both x↑.hd and x↑.tl.

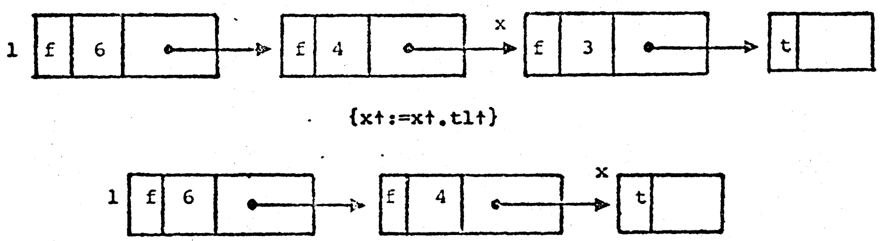
Now the program to delete odd numbers from l is reasonable.

```
x:=l;
while ¬x↑.empty do
     if odd(x↑.hd)
           then   x↑:=x↑.tl↑
           else   x:=x↑.tl
```

Lest the reader suspect this example was cooked, several more are given in an appendix.

Figure 2. Two methods of list representation

(a) pointer swing



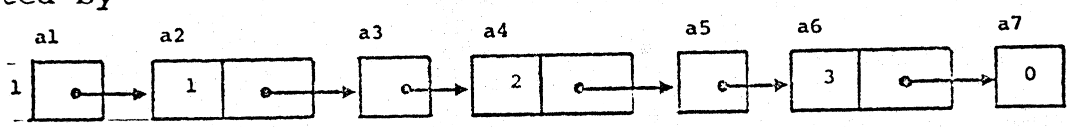
(b) node overwrite

Recommendations for Language Design and Implementation

The language should allow people to use the node overwriting strategy and should not penalize them excessively by implementing structures naively. Although PASCAL and ALGOL-68 allow it I suspect their implementors discourage it.

An Implementation

The most straightforward implementation (used in CPL) uses extra pointers which the user cannot access directly. For example the list l= (1,2,3) is represented by



The addresses of the smaller boxes represent pointer values; assignments through pointers change the contents of these boxes. There is nothing the user can say to change the contents of the larger boxes. In terms of PASCAL notation

```
l=a1

l↑=a2

l↑.tl=a3

l↑.tl↑=a4

etc.
```

However, recall that

```
            l↑.tl := p

means       l↑ := <l↑.hd,p>
```

which changes the contents of a1, not a2.

The reader's reaction to this description is likely to be what mine was: "Hiding all those pointers from the user is a bad thing." Considering how long it took me to reject that view, I doubt anything I say can decisively refute it. My only suggestion is that he try to write some of the example programs using the pointer swinging strategy, and then multiply the hassle he experiences by the number of programmers who will write similar programs.

Appendix: Further examples of node overwrite programs

Each of the examples is done using the node overwrite strategy. I found the pointer swinging versions troublesome.

(a) List insertion.

Using the node overwrite definition of list, insert i in the ordered list l.

```
procedure insert (i:integer; l:list);
     var n,x:list;
     begin x:=l;
           while ¬x↑.empty & i<x↑.hd do x:=x↑.tl;
               new (n); {allocate a new node}
               n↑:=x↑;
               x↑.hd:=i; x↑.tl:=n; x↑.empty:=false
     end
```

(PASCAL's syntax would be improved if one could replace the last line by something like

```
x↑:=<i,n>)
```

(b) Tree insertion.

Given

```
type tree=↑ record case empty:Boolean of
                         true:  ;
                         false:(data:integer;l,r:tree)
           end
```

write a procedure to insert into a tree so that post-order scan orders the numbers.

```
procedure insert (i:integer;t:tree);
     var x:tree;
     begin x:=t;
           while ¬x↑.empty do
                x:=if i<x↑.data
                         then x↑.l
                         else x↑.r ;
           x↑.empty:=false;
           x↑.data:=i;
           new (x↑.l);  x↑.l↑.empty:=true;
           new (x↑.r);  x↑.r↑.empty:=true
     end
```

This example illustrates a potentially disastrous waste of space caused by the node overwrite strategy. The leaves of the tree are always empty yet must be big enough to hold an integer and two pointers; thus a tree requires twice as much space as it should. A minor re-design of PASCAL might allow the implementor to be clever and materialize empty nodes only when there are multiple references to them.

(c) Radix Sort.

```
procedure sort (n:list);
     var f,l: array[0..9] of list;
         c,t: integer;
     {assume all the numbers are <100000}
     begin for t:=0 to 9 do new (f[t]);
           c:=1;
           while c<100000 do
             begin for t:=0 to 9 do l[t]:=f[t];
                   while ¬n↑.empty do
                   begin t:=n↑.hd/c mod10;
                         l[t]↑:=n↑;
                         l[t]:=n↑.tl;
                           n:= n↑.tl
                    end;
                    for t:=9 downto 0 do
                    begin l[t]↑:=n↑; n↑:=f[t]↑ end
                   end
          end
```

The pointer swinging approach will require one to worry about empty lists; here one only has to be sure to concatenate from back to front.

Another apparent expense of node overwriting is brought out by this example. Suppose the hds of lists were 80 character arrays. Then assignments like l[t]↑:=n↑ might involve many memory references. The implementor can ameliorate things by using pointers behind the scenes. He should resist the temptation to allow the user to swing these pointers.

(d) Two-way lists.

Node overwriting seems inappropriate for two-way lists. The same declaration as for tree will suffice for nodes on two-way lists. To delete a node x from its list one <u>could</u> say

```
t:= x↑.r;
x↑:= x↑.l↑;
x↑.r:=t
```

but that seems strange and wouldn't work for two node circular lists. A pointer swinging change

```
x↑.l↑.r:=x↑.r;
x↑.r↑.l:= x↑.l
```

seems better.

(e) Expression evaluation.

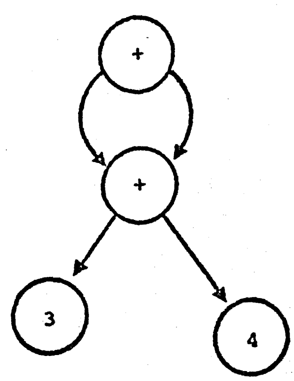
Suppose arithmetic expressions are represented according to

```
type exp=↑record case op:etype of
                   const: (val: integer);
                   sum: (l,r: exp);
          end
```

The following procedure evaluates the expression, avoiding re-evaluation of shared sub-expressions.

```
procedure eval(e:exp);
     var t:integer;
     begin if e↑.etype=sum then
           begin eval(e↑.l);
                 eval (e↑.r);
                 e↑.etype:=const;
                 e↑.val:=e↑.r↑.val+e↑.l↑.val
           end
```

The pointer swinging version of this program would involve assignments like e↑.l:=v and e↑.r:=v. Aside from being clumsier it would have to perform three additions instead of two on a structure like



References

[S] Strachey, C., CPL Working Papers, University of London Institute of Computer Science, 1966.

[P] Park, D., Some Semantics for Data Structures, Machine Intelligence 3, pp. 351-371, American Elsevier, 1968.

[vW] van Wijngaarden, et.al., Draft Report on the Programming Language ALGOL-68, Mathematishe Centrum, Amsterdam, 1968.

[W] Wirth, N., The Programming Language PASCAL, Acta Informatica 1,1, 35-63 (1971).

Distribution

BAUDELAIRE, Patrick
BOBROW, Dan
DEUTSCH, Peter
ELKIND, Jerry
FIALA, Ed
GESCHKE, Chuck
GUIBAS, Leo
HEWITT, Carl
JEROME, Suzan
KAY, Alan
LAMPSON, Butler
MCCREIGHT, Ed
MITCHELL, Jim
MORRIS, Jim
SATTERTHWAITE, Ed
SIMONYI, Charles
STURGIS, Howard
SWEET, Dick
TAYLOR, Robert