On Plex Bases in SETL

Stefan M. Freudenberger

SETL Newsletter 216
Courant Institute of Mathematical Sciences
New York University

17. May 1981

In ALGOL 68, a linear linked list can be introduced by the following declarations:

```
mode element   = struct(amode value, ref element next);
mode list      = struct(ref element first, last);
```

We can then obtain a (still empty) list by declaring:

```
list mylist := (nil, nil);
```

To generate a new element and to insert it into an actual list, we define the following procedure:

```
proc generate = (ref list list) void :
                (heap element new;
                   if ref element (first of list) is nil then
                      first of list := new;
                   else
                      next of last of list := new;
                   fi;
                   last of list := new;
                   next of new := nil
                )
```

In SETL, we can declare the same structure as follows:

```
init
    value := {}, next := {}, first := {}, last := {};
repr
    plex base element;
        value:          local smap(elmt element) amode;
        next:           local smap(elmt element) elmt element;
    plex base list;
        first:          local smap(elmt list) elmt element;
        last:           local smap(elmt list) elmt element;
end repr;
```

If we now declare:

```
mylist: elmt list;
```

and initialise it by executing:

```
mylist := newat;
```

then we have the same structure MYLIST as defined before. (Alas, SETL does not permit to merge a variable's mode declaration and its initialisation, as ALGOL 68 does.) The corresponding procedure for generating a new element and inserting it into an actual list is as follows:

```
procedure generate(list);
    repr list: elmt list; new: elmt element; end repr;
    new := newat;
    if first(list) = om then
        first(list) := new;
    else
        next(last(list)) := new;
    end if;
    last(list) := new;
end procedure generate;
```

where we assume that a programme calling GENERATE would also supply the required declaration for the procedure:

```
generate: procedure(elmt list);
```

The idea in this representation is as follows. An element of a linked list itself does not correspond to any computational value. Indeed, we only would like to create such an element (and assure its uniqueness), and to compare two elements for equality. If we don't need an element any longer, we just discard it and let the execution environment deal with the clean-up problem.

The SETL atom value represents a value with just these properties. It then comes natural to represent the fields of the list elements as maps from atoms to the field values. In pure SETL, this would have been done as shown above, but without the data structure declarations given. This would have resulted in a structure which allows far more general operations than we would expect to do with a linked list. It would permit, for example, to print the VALUE's of all list elements by just writing:

```
print(value);
```

For a linked list, we would have expected to explicitly state how, given the current element, we can find the next element. From this experience, one might suspect that the SETL system maintains more information about each element of a list than is required. Obviously, too, we shall have to pay, both in space and execution time, for maintaining this information. This is where the SETL plex base data structure is useful.

A SETL Plex Base is a very static data structure, and has several semantic restrictions. Most notably, it prohibits all operations which require a hashing operation or an iteration. As a consequence, this data structure does not have to maintain a hash table, nor link pointers between the individual elements of the structure. Thus it is more space efficient, and since this information does not need to be maintained, also more time efficient.

A Plex Object is a set mode based locally onto a plex base. Only local basing is allowed for plex bases. Since iteration over a based object requires an iteration over its base, it is not possible to iterate over a plex object. Since we cannot perform a hashing operation on a plex base, we can only transfer pointers to elements of the base, but can never compute the address of a plex element block, given its value. This leads to the following operations defined for plex objects:

If S is a local set of elements of a plex base, and X is an element of the same plex base, then the following operations are legal:

```
s with:= x                    $ add X to S
s less:= x                    $ delete X from S
x ∈ s, x ∉ s                  $ test whether X is an element of S
```

If F is a local map from elements of a plex base, and X is again an element of the same plex base, then the following operations are legal:

```
f(x), f{x}                    $ return the value of F at X
f(x) := ..., f{x} := ...      $ assign the right-hand-side to F at X
f lessf:= x                   $ delete the range of F at X
```

If X and Y are both elements of the same plex base, then we can assign Y to X:

```
x := y                        $ pointer assignment
x := newat;                   $ generate a new plex element
```

As mentioned before, it is not possible to (implicitly) iterate over a plex object. This implies that it is not possible to (implicitly) copy a plex object, and together with the static nature of locally based objects in general, this means that plex objects can not be used in an assignment statement. Note that in the above lists, we required all operations to be update operations, e.g. S **less**:= X, and did not allow the expression form of the operator, i.e. S **less** X. Moreover, this implies that plex objects must be initialised using an **init** statement in the scope header of the scope in which they are defined. This scope must be a static scope, i.e. cannot be a procedure scope, since otherwise the SETL semantics requires that the object be re-initialised every time the scope is entered. Thus plex objects can only be declared in a library header, a directory, a module header, or a programme header. Finally, plex objects cannot be used as actual parameters, since the semantics of the value transfer to the formal parameter is equivalent to the semantics of an assignment.

There is one last item which we believe deserves mentioning: ALGOL 68 allows **ref amodes** to appear freely in union modes, as long as the enumerated modes cannot be equivalenced using de-reference and de-procedure operations. In its current definition, SETL does not allow us to do so. More precisely, let X have the mode **elmt plex_base**, and let Y have the mode **general**, the SETL equivalent to the ALGOL 68 **union** mode. The even though the assignment Y := X would be legal, the inverse assignment X := Y is not, even if, at execution time, Y would always have a value with an **elmt plex_base** mode.