SETL Newsletter # 174

Robert Dewar
July 19, 1976

Relaxation of Basing Restrictions.

The current scheme for basings involves some significant restriction on programs using basings and implies the existence of time and space consuming tests to ensure that these restrictions are met. For example

```
DCL     a : base;
DCL     b : smap (∈ a) int;
DCL     c : ∈ a;
.
.
.
b(c);
```

involves testing a to see if it has been diminished and c to see if its corresponding base block has been deleted. This newsletter proposes relaxing these restrictions and discusses the implementation consequences.

Language changes

In the language as proposed, basing declarations would be purely advisory;and no harm, except possible unnecessary space (more rarely time) wastage, would result from 'bad' declarations. Even the following program is legal:

```
DCL     a : base;
DCL     b : smap (∈ a) int;
DCL     c : ∈ a;
a = nl;
b = {< 1,2>, <3,4>, <5,6>};
read (c);
print (a, b(c));
finish;
```

this program would print nl followed by 2, 4, 6 or om depending on the value of c. Note that as far as the programmer is concerned the basing "violations" have no effect, in particular a is always nl.

The only remaining restrictions are those concerned with the passing of bases to routines. These restrictions will not be removed.

Another addition to the language allows the extended use of virtual bases. In the sample program given, a has no function except to serve as a common base for b and c in their declarations. Such a base may be declared to be virtual. A variable declared to be a virtual base may only be used as a base in other declarations, no other uses are legal.

Implementation

If an object x based on a base y is assigned a value which implies the addition of an element to the set y, then the element will actually be added to the set, but marked as deleted. This deleted element carries an index and a local map block like any other element of the base, but is invisible to operations on y itself.

Thus each base really is two values: the value expected from SETL semantics and visible to a SETL program, and a superset value large enough to maintain the validity of all basings. This secures the advantages of the basing without introducting dreaded 'pointer semantics'.

A potential problem is that sets get cluttered with obsolete basing garbage which the program cannot remove. An analog of the problem is the possibility of the streets of New York being covered with invisible garbage which cannot be removed even after a strike is over.

The solution to the invisible garbage problem is to trace all blocks which are deleted and not referenced by any remote map. This sounds difficult (and is difficult) but it is no worse than the problem of compacting indices, which is essential in any case under the original scheme. The basic approach is as follows:

1) Link all remote maps to the base block by pointer reversal in pass 1.
2) Construct a bit map (indexed by element index values) which shows which blocks are referenced by remote maps (or alive and well in the base).
3) Compact indices on the basis of the bit map of (2). Adjust remote maps accordingly.
4) Remove all elements in the set whose index is not marked in the bit map.

The bit map manipulations are similar to those used by the main storage compaction scheme so code can be shared.

Since the loading of the table is known, this procedure need only be carried out for sets with many deleted elements.

Execution Advantages

Since basings are always "true", no checks need be made which reduces the space and time for many important code sequences.

Optimization Advantages

The optimizer need only guess relationships, not guarantee them. For example if the construction

f(x)

appears and f is known to be a map, then declaring f, x to be based on some virtual base is unlikely to substantially slow down the program (no matter what it is), but may well give a substantial speed-up.