SETL Newsletter # 158

E. Schonberg

Implementation of Reference Counts in the SETL System.

October 15, 1975

This Newsletter extends the remarks of SETL Newsletter no. 57 (Hank Warren: copy minimization in SETL) and describes a somewhat restricted implementation of reference counts (RF's) in the SETLA interpretive system.

Reference counts are used to suppress redundant copy operations which arise logically in SETL whenever an assignment implying a sharing of values between program variables is performed. [Till now our implementation has inserted automatic copy operations whenever a composite object was retrieved from, or made part of, another composite structure. The resulting overhead was sizeable, as it involved not only the copying routines but forced repeated garbage collections--which are expensive in the current system (~ 2 sec for 32000 words of dynamic storage).]

In the improvement just implemented, reference counts are attached to every multiword heap block which may be the value of a program variable or temporary. The value of the reference count of a heap block must at all times be equal to, or greater than the number of live pointers referencing that block. These pointers are located either in root words of long items, or in the clash-lists of hash-table entries.

In the absence of live-dead analysis, a root word must be considered **live** if it is either a symbol table entry, a local variable of a currently active procedure, or one of the calling parameters of a currently active procedure. Operations which perform assignments to any of the above must therefore modify reference counts appropriately.

Potentially destructive operations, i.e. operations which at the implementation level destroy one of their arguments (e.g. set union, which is performed in place whenever possible) must examine the reference count of their arguments, and perform a copy if this RF is greater than zero.

Sinister assignments, e.g.

```
f(x) = y;
```

imply that the value of f, or a subpart thereof, is to be modified. This can be done in place if the value of the variable f is not shared by others, i.e. if the RF of set f is no greater than one. Otherwise f must be copied before modification. Generalized sinister assignments, when implemented along the lines described in On Programming pp. 181 ff can be treated in the same fashion. We discuss them in some detail below.

Before describing any additional details of the current implementation, let us note that the scheme as described is safe but not minimal, in the sense that all necessary copies are detected, but that some additional redundant copying is still performed. The savings over previous implementations are nevertheless substantial.

The following types of operations must modify reference counts: assignments, sinister assignments, creation of composite objects, and destructive operations. We discuss each operation type in detail.

Assignment statements

The statement: x = y; (1)

is handled in a manner determined by the binding of x, and the namescope in effect. 3 cases must be considered:

a) x is global (to the whole program, or to a namescope). In this case, (1) is executed as follows:

a1.-The RF of the old value of x is decreased by one

a2.-The RF of y is increased by one.

a3.- The rootword of y is placed in the entry for x.

b) x is local to a procedure. In this case (1) is executed as above. In addition, if the variable is not declared owned by a procedure, it is dead when the procedure is inactive, and its reference count may be decreased on exit from it.

c) x is one of the arguments of the currently active procedure. In this case, while the procedure is active, the same remarks as above apply. However, as SETL allows procedures to modify their parameters, the RF's of program variables are not affected by becoming the calling parameters of a procedure P. On exit from P, the delayed argument return mechanism will have to replace the (possibly updated) value of each argument, in the environment in which it resides (symbol table, environment block or parameter list of calling procedure). Updating of RF as described by (a-c) above will be carried out for each argument reassignment.

A slight irregularity in our scheme becomes necessary at this point. If a program variable x appears as one of the arguments of a call *proc* (x,y), the RF of x is not incremented at the point of call. If x is reassigned within *proc*, the RF of the old value should therefore not be decremented. Successive reassignments of x will therefore produce RF's which are greater than necessary. This seems of little practical consequence, as calling parameters are seldom reassigned, and almost never reassigned successively within the same procedure.

Sinister assignments

Sinister assignments can modify their left hand side (the target of the assignment) in place, as long as its RF is no greater than one. The RF of a newly modified block is set to zero, and is subsequently incremented to 1 when placed in the symbol table. Sinister assignments must also modify the RF's of their right-hand sides. The following cases arise:

a) f(x) = y;

The RF's of x and y are incremented by 1, because they will become subparts of f.

b) $F(x_1, x_2 \ldots x_n) = y$;

the RF of y must be incremented. Those of $x_1, \ldots x_n$ will be incremented when the argument tuple is built.

c) f{x} = y;

The RF's of x and of each member of y (which must be a set) are incremented.

d) $f\{x_1 \ldots x_n\} = y$;

Same as c.

e) f[x] = y;

The RF's of each member of x and y (which must both be sets) are incremented.

f) $f[x_1 \ldots x_n] = y$;

Same as e, for all members of $x_1 \ldots x_n$ and y.

The operator in can be treated as a special case of a): the new member of the set being augmented has its RF incremented.

A similar rule applies to the creation of composite objects. The RF's of all items about to become members of sets or components of tuples are incremented by one. At the implementation level, the procedures *augment*, *gentup*, *tuppadd1* do the necessary incrementing. This rule is safe but not minimal, as the modification or creation operation in question might be applied to a compiler temporary (which has an RF of 0). However, the rule does take care of the case when an expression is subsequently assigned into a program variable.

Destructive operations

These operations include +, - and / in their various meanings. At the implementation level they are performed in place. In the absence of live-dead analysis, their first argument must be copied if its RF is different from zero. Before copying the RF of the old value must be decremented by 1, as long as it does not become zero. This again is a safe but non-minimal prescription.

Generalized sinister assignments

The statement $\underline{op}_2\underline{op}_1 x$ = expr; (2)

is implemented by the following expansion:

$$t = \underline{op}_1 x; \quad (3)$$

$$\underline{op}_2 t = \text{expr}; \quad (4)$$

$$\underline{op}_1 x = t; \quad (5)$$

Line (3) retrieves a pointer from x and assigns it to the compiler temporary t. Line (4) performs a simple sinister assignment on t. The safety of this operation obeys the rules stated before. In the simplest case we have $RF_x = 1$, and the RF of $\underline{op}_1 x$ is also 1. (4) can then be executed in place. This of course means that the assignment to t appearing in line (3) has not modified the RF of the block being assigned into it. Finally (5) can also be performed in place. If x has become a subpart of other structures, $RF_x > 1$. However $\underline{op}_1 x$ may still have an RF of 1.

In this case, (4) will be performed in place, while (5) will require copying. This, however, is unsafe, because (4) is actually executed within x, and will therefore modify its value before the copying triggered by (5) is executed. The same problem can arise with longer sequences of storage operators. The source of the difficulty is the possible imbalance that may develop between the RF of a composite and those of its subparts. The straightforward (and expensive) solution is to update the RF's of all subparts (to any depth) whenever the RF of a composite is modified. This however seems unacceptably expensive. From the preceding example it is clear that RF's of subparts need to be updated only immediately before the points at which they may be modified, at which time they can be set to the RF of their parent composite object. This need only be done for the code sequences produced by generalized sinister assignments.

Finally, copying itself can be restricted to a single level, as long as the RF's of subparts are incremented by 1. This means that only the hash tables of sets need be copied, and the RF's of set members updated. This will always be cheaper than copying to full depths, and is clearly safe. Decrementing the RF's of subparts of dead composites (e.g. old symbol table entries before a reassignment) is probably a refinement that lies beyond the point of diminishing returns.