SETL Newsletter 57

Minimizing Copying in SETL: Preliminary Observations

October 8, 1971
H. S. Warren

In general, the evaluation of an expression such as u op v can be done most economically if the context permits

(1) destroying either u or v, or
(2) working only with references to u and v.

For example, if S is a set and "a" is a long item (long string, set, tuple, etc.) then the most efficient way to evaluate S with a is to destroy S by placing in it a reference to "a". As another example, if "j" is a long integer, then the most efficient implementation if "i = j + 1" is to destroy j by adding 1 to it, and then to make i reference the result itself, rather than a copy of it.

A first-order optimization of SETL operations by minimizing copying might be:

(1) If a term in an expression is dead, then it may be destroyed.
(2) Do not use references: always generate fresh copies of items for operations such as with and assignment.

Here a "dead" term is a term that is not directly followed by a use, for any program path from the point in question. "Marring" occurrences may be skipped over.

This first order optimization of copying is entirely static, as dead terms may be determined by the compiler. In essence, the compiler looks ahead through all possible program paths.

When the compiler compiles a call to a routine such as WITH or PLUS, it could pass to the routine a live/dead flag for each operand. Alternatively, separate routines could be used.

With first order optimization, the statement "S = S with x" will be executed without copying S. To see this most clearly, it helps to expand it in terms of a "compiler temporary," which is understood have have only local significance:

```
(1)  temp = S with x;
(2)  S = temp;
```

At statement (1), S is dead, and therefore the WITH routine may destroy it by simply putting into it a copy of x. The assignment to a compiler temporary never requires a copy operation.

Regarding assignments, a simple assignment such as A = B is treated as follows. If A is dead, the assignment need not be done (and would presumably be deleted by the compiler). If A is live and B is dead, the assignment may be done by making A reference the same structure that B references. Only if both A and B are live need a copy be done. In statement (2) above, "temp" is certainly dead, so the assignment may be done without copying.

The next level of minimizing copying to be considered is to work with references to "long" items whenever possible. That is, in "S with x", we wish to avoid copying x. The compiler can determine the obvious cases where x is never changed after the point in question, in which case working with references is safe. For example, if x is a constant, as in S with 'abcde', it is certainly not necessary to make a fresh copy of the string 'abcde'.

This idea can be fully exploited by always using references to long items (the short items might as well be copied). It is then necessary to maintain, at run time, a "reference counter" associated with each long item, so that it can be determined when it is safe to modify a value without copying it.

That is, this "second order" of optimization by minimizing copying operations is governed by the following rules:

(1) If a term in an expression is dead and its reference counter is 1, then it may be destroyed.

(2) Always use references to long items when they are put into aggregates or enter into an assignment.

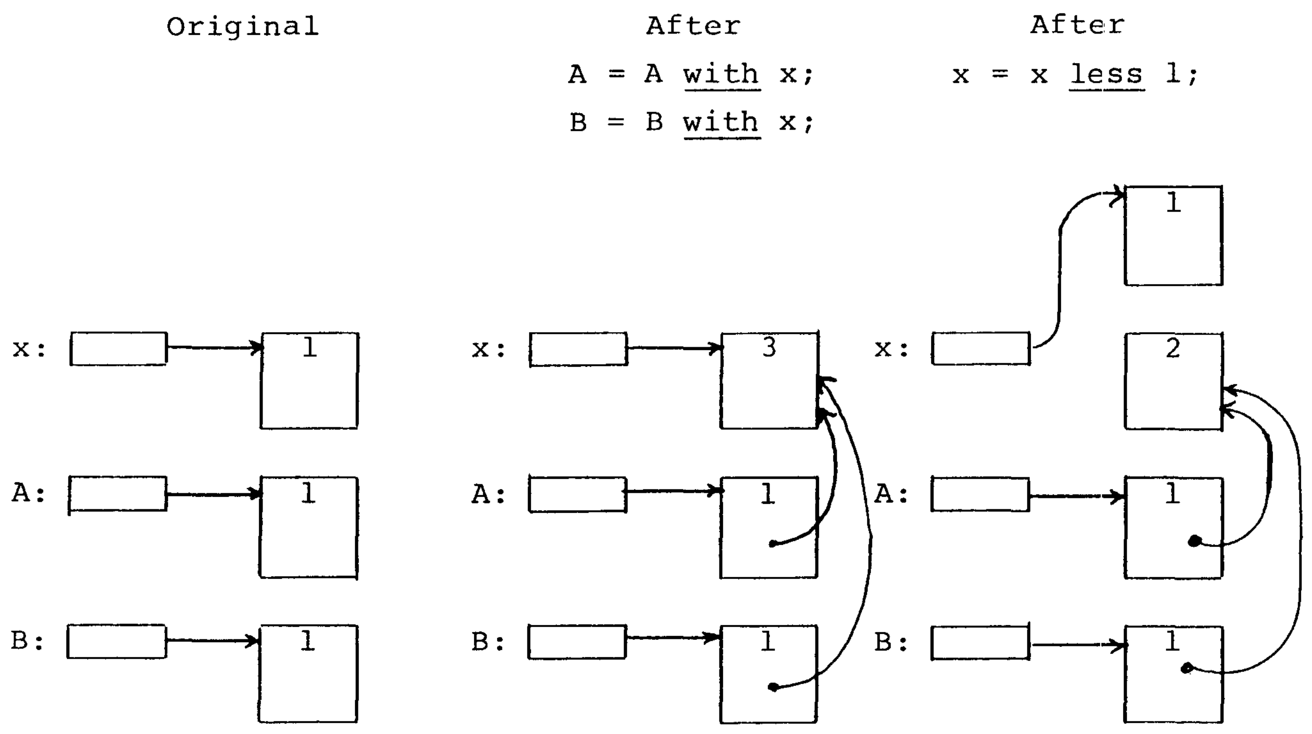
To get the benefits of (2), we have paid a price in (1). However, the benefits must always be at least as large as the price.

Consider the following SETL code, where all variables are sets:

```
(1)    A = A with x;
(2)    B = B with x;
(3)    x = x less 1;
```

The first order optimizer will copy x at lines (1) and (2). The second order optimizer will copy x only at line (3).

It is helpful to have a mental picture such as the one below, of the processes involved. The SETL variables x, A, B can be thought of as names for one-word storage locations which



point to data structures in the storage heap when their values won't fit in the "root word". The numbers in the data structures of the above figure are the reference counts.

One can imagine higher orders of optimization by minimizing copying. For example, for S = A with x (with A live) one could partially destroy A by somehow putting into it a specially marked "x" which is interpreted as being present for S but not for A. As another example, there might be cases where one could destroy a set and later restore it. That is, the statement "X = A - (B with x), with A and B live, might be executed as:

```
          if x ∈ B then X = A - B;
          else augment(B,x); X = A - B; diminish (B,x);;
```