SETL Newsletter # 175

S.C. Liu
E. Schonberg
July 21, 1976

More on Copy Optimization.

SETL newsletter # 164 remarks that the copy instruction required by the value semantics of SETL can be inserted either on assignment, or upon modification of a composite object; and that copying only on modification as done in the current system and as implied by the destructive use condition (see O.V.H.L.I) can be inefficient in some cases. The example:

```
A = B; (∀ x ∈ s) A(x) = f(x);;          (1)
```

makes this plain:if A is copied before modification, it will be copied (#s) times within the loop, unless some dynamic scheme (reference counts or shared bits) indicates to the processor that only the first copy is required. However, if we write:

```
A = B; (∀ x ∈ s) A = copy(A); A(x) = f(x);;          (2)
```

then it is clear that the copy operation can be moved out of the loop:

```
A = B; A = copy(A); (∀ x ∈ s) A(x) = f(x);;          (3)
```

If a static analysis can determine safe instances of copy motion, then we can retain the analysis of OVHL, insert copies only on modification, and perform the safe code motion in a separate optimization pass.

Let us consider again the code sequence (2):

```
L1:     A = B;
L2:     (∀ x ∈ s)
L3:     A = copy(A);
L4:     A(x) = f(x);;
```

Note first that a dynamic system for copy optimization will perform a conditional copy at L3, based on the shared-bit of A_{i3} (i.e. the ivariable occurrence of A at line L3). It is only on the first pass through the loop that this bis is set, and the copy must be performed. A static analysis can recover this result by noting that within the loop, A does not become part of any other live variable, so that the copy made on the first pass and subsequently modified at L4 is a stand-alone value, modifiable in place. This is the condition we want to formalize.

Another remark: the code fragment (3):

```
L1:     A = B;
L2:     A = copy(A);
L3:     (∀ x ∈ s)
L4:     A(x) = f(x);;
```

is easier to analyze: A_{i4} can be used destructively because

$$\text{crpart}^{-1}\ [\text{crthis}(A_{i4})] = \{A_{02}, A_{04}\} \quad .$$

and both these values are dead before executing L4. (The copy at L2 means that $\text{crthis}(A_{02}) = \{A_{02}\}$). Because of the expense of calculating the mappings crpart, etc., it is out of the question to insert a copy outside of the loop for a variables which are modified within it, and then verify whether the inner occurrences of these variables still satisfy the destructive use condition. Rather, we want to perform a static analysis on the original code (2), with a copy assumed on modification.

An exact condition for copy motion.

The destructive use condition tells us that if the ivariable a_i must be copied, then the set $\text{crpart}^{-1}\ [\text{crthis}(a_i)] \neq \underline{n\ell}$, and its elements are not dead at the point of destructive use (i.e., at the point where a copy becomes necessary).

The condition we seek depends on the elements of crthis(a_i) and in particular on whether a_i is a value created inside or outside the loop.

a) If a_i is (a pointer to) a value acquired outside the loop, then on entrance to the loop several variables refer to that value. If the copy is performed before entering the loop, the variable a point to a new (unshared) block. It can be used destructively within the loop if it is dead at the point of use, regardless of the fate of the outside variables which transmitted a value to it. For example, in

```
a = b; (∀ x ∈ s) a(x) = f(x);; b with z;
```

a can be copied outside the loop (it is dead before executing a(x) =...) while in the code fragment:

```
a = b; (∀ x ∈ s) t = a with f(x);;                    (4)
```

a is live at the point of destructive use, and the copy cannot be moved.

b) If a_i is (a pointer to) a value acquired within the loop, then each iteration through the loop produces a new pointer (not necessarily distinct from its value on previous or subsequent iterations) and the full destructive use condition must be applied to these values, i.e., they must not have become part of any live object between the time they were created and the point of destructive use.

Let us designate by Lp the set of instructions within the loop, and by instr(o) the instruction which creates the ovariable o. We then define:

```
crthis_in(i) = [+: o ∈ ud(i) | instr(o) in Lp] crthis(o);
```

The copy motion condition can then be written as:

$$\text{move(i)} = \text{live } (\text{crpart}^{-1}\ [\text{crthis}_{in}(i)] + \{\text{ovar(i)}\})\ \underline{\text{eq}}\ \underline{\text{nl}}$$

Note that ovar(i) must appear explicitly in this expression, to cover the case where $\text{crthis}_{in}(i)$ is empty, as in (4) above.

Some simple examples.

a) a = b; (∀ x ∈ s) a with x; c with a;;
the move condition is false, because a is live at the point of destructive use.

b) a = b; (∀ x ∈ s) a with f(x); b less g(x);;
the move condition is true for both a and b.

c) a = b; (∀ x ∈ s) a(x) = f(x).... a = t(x);; $ t is a map.
The move condition is false because $\text{crthis}(a_i)$ at the point of copy includes t(x)

d) a = b; (∀ x ∈ s) a(x) = f(x)... a = g(x);;
$ g is a function.
The move condition cannot be ascertained without inter-procedural analysis, because the function g might return a shared object.