SETL Newsletter # 212

J. Schwartz
September 30, 1979

A Note on Program Genesis

Where subtle algorithmic inventions are not in question, the process of programming is experienced as having a straightforward manipulative character (somewhat like algebraic manipulation, but perhaps even less dependent on strict use of a formalism.) It is interesting to speculate on the reason for this, and to attempt capture, in some semi-formal 'ultra-high level' language, of the implicit structures which guide the elaboration of programs at SETL-like levels. If such a language can be outlined clearly enough, it might become possible to use it systematically as a language of specification for the SETL programmer, or even to execute some useful subpart of it.

As a central statement of the desired ultra-high level language, we propose

(1) force C keeping V,

where C is boolean condition depending on the variables of a program, and where V is some other condition, or more generally some combination of conditions and values to be preserved. Semantically, statement (1) is a (highly nonprocedural) directive to modify the values of variables in such a way that C becomes true, but without changing V. Any predicates comprised in V are assumed to be true immediately before (1) is executed.

Statements of the form (1) can be compounded. To see this, suppose for simplicity that V is a predicate. Then after the execution of (1) the condition C & V is true; thus if (C & V) $\rightarrow V_2$, we can embed (1) in the longer program

(2) force C keeping V; force C_2 keeping V_2;...

and C_2 & V_2 will then be true after the second statement of (2).

Thus (2) may be considered to arise from a chain of implications

(3) $$(C \ \& \ V) \rightarrow V_2,\ (C_2 \ \& \ V_2) \rightarrow V_3, \ldots,\ (C_k \ \& \ V_k) \rightarrow V_{k+1},$$

where V is an assumed input condition and V_{k+1} is a desired output condition.

A statement (1) can be 'expanded' or 'implemented' in various ways, leading to various familiar program structures. Suppose that the variables of C and V fall into two classes a and b. Then (1) can be expanded as

(4)
```
(while not C)
    ā := a; b̄ := b;
    a := α(a,b);
    force V(ā, b̄) = V(a, b) keeping [ā,b̄];
end while;
```

Here α can be any expression, but should be chosen to advance the loop (4) to termination. (The force statement imbedded within the preceding loop is formulated in the manner appropriate for a function V; if instead V is a predicate it should be

force $V(\bar{a}, \bar{b}) \rightarrow V(a, b)$ keeping $[\bar{a},\bar{b}]$

instead.) This imbedded force statement can of course be implemented in any desired fashion, perhaps by an imbedded loop which might force $C'(\bar{a}, \bar{b}, b)$ keeping $[\bar{a}, \bar{b}]$, where $(C'(\bar{a},\bar{b},b)$ & $V(\bar{a}, \bar{b})) \rightarrow V(\alpha(\bar{a},\bar{b}), b)$, or perhaps more simply by an assignment $b := \beta(a,b)$, where

(5) $$V(a,b) \rightarrow V(\alpha(a,b),\ \beta(\alpha(a,b),b)).$$

In this last case the auxiliary variables $\bar{a},\bar{b}$ will be dead and need never appear in the loop (4) at all.

Note that the manner in which we have written the expansion (4) of (1) captures an interesting distinction, which any self-observant programmer will have noticed: that between 'primary' or 'motive' statements, which move loops like (4) in a direction leading to termination, and 'compensatory' statements or 'adjustments' which, by repairing the damage caused by a 'primary' statement, restores the value or condition V. (Adjustments arising in set-theoretic strength reduction are of course a prime example of this.) We consider in this connection that a major if humdrum part of programming skill is wide enough knowledge of such 'adjustments' to cover most cases arising in common practice.

A particularly easy case is that in which the variable b of (4) is absent and the assignment a := α(a) itself preserves V. An example of this is bubblesort, which we can write as

```
(while  ∃n: in [2..#f] | f(n-1) < f(n))
        [f(n-1), f(n)] := [f(n), f(n-1)];
end while;
```

Conditional constructions will arise if one of the parts of V (say for simplicity V itself) is a disjunction, since

(6) force C keeping (V_1 or V_2)

can expand as

(7)
```
if V1 force C keeping V1
elseif V2 force C keeping V2.
```

The two force statements appearing on the two branches of this if can of course expand either into simple statements or into further loops, etc. Of course, they may expand quite differently.

A higher level form in which expanded loops like (4) might conceivably be written is

(8) force C keeping V by a := α(a, b)

Here we mean to imply that only the 'motive' statement need be given explicitly, the 'adjustments' then being supplied automatically. In this sense we might then express the essence of bubblesort by writing something like

(9) force (∀n in [2.. #f] | f(n-1) ≥ f(n))
keeping range f by [f(k-1), f(k)] = [f(k, f(k-1)].