SETL Newsletter 159

J. Schwartz
November 6, 1975

On the 'Base Form' of Algorithms

1. Introduction, Examples.

The 'base form' of an algorithm is that 'simplest' representation which it can be given if it is written in a way deliberately excluding all 'non-creative' or 'routine' optimizations; we mean to exclude optimizations even if they lie beyond the range of present-day automatic optimization technique, provided only that the manual application of these optimizations is a truly routine matter not involving any invention at the mathematical level. Of course, optimizations of this 'essentially routine' class should ultimately become amenable to automatic treatment. Among the optimizations to which we allude are almost all matters related to data structure choice, procedure integration, recursion removal, 'formal differentiation' (which Earley calls 'iterator inversion'), conversion of programs using various useful non-standard control structures (such as backtracking) to programs envolving standard control structures only, use of 'memo functions' (cf. NL 155), etc.

One rough but plausible way of describing these optimizations is to say that they can be characterised in a few words to a skilled programmer, who can then apply them with little doubt as to what is meant. Of course, the variant of any particular algorithm which we call its base form will change as our understanding of high level program structure and optimization becomes more profound. Moreover, an algorithm's base form will depend on the class of operations which we are willing to regard as 'primitive'.

An algorithm written in base form can and should make use of programmed subprocedures or macros where these clarify its logical structure.

The following examples of algorithms written in base form will clarify what is meant.

(a) Bubble Sort of a vector v:

```
(while 1 ≤ ∃n < # v | v(n) > v(n+1)) <v(n+1);v(n)> = <v(n), v(n+1)>;;
```

The more conventional form of bubble sort is obtained from this by turning the existential into a search loop, and by noting that during search the range of indices within which an ill-ordered pair can exist will only decrease when a swap is performed, and then only by 1. This is essentially a type of formal differentiation (as applied to vectors rather than sets.)

(b) Heap sort of a vector v:

```
(# v ≥ ∀n > 1) makeheap(c,n);  <v(n),v(1)> = <v(n),v(1)>; end ∀;

definef makeheap(v,n);

(while 1 < ∃m ≤ n |v(m/2)<v(m))  <v(m),v(m/2)> = <v(m/2), v(m)>;;

return;
end makeheap;
```

This is optimised by noting that on all but the first (outer) iteration, there can be at most one m/2 such that v(m/2) > v(m); to optimise the first iteration we keep track of the m for which v(m/2) > v(m) can hold (this is a type of formal differentiation.)

(c) Transformation of a grammar to Greibach normal form.

A context-free grammar G is given as a set of pairs p with two components ℓhs and rhs, where ℓhs is a symbol and rhs a tuple of symbols; *intsymbs* denotes the set of all intermediate symbols which appear as left-hand sides.

One way of transforming G to an equivalent grammar is to take some production $r \to \ell_1 \ell_2 \ldots \ell_n$ of G such that $\ell \in$ intsymbs, and to replace it by all the productions $r \to m_1 \ldots m_k \ell_2 \ldots \ell_n$, where $\ell_1 \to m_1 \ldots m_k$ belongs to G. Another transformation is as follows: if $r \in$ intsymbs and G contains productions $r \to r\,\alpha$ and $r \to \beta$, where α and β are strings of symbols, all the terminal strings generated by r are all generated from the class of strings which are either a β or a β followed by a sequence of α's. Hence we can replace the productions whose <u>ℓhs</u> is r by the following productions, in which r' is a new intermediate symbol: $r \to \beta$, $r \to \beta r'$, and $r' \to \alpha\, r'$, $r' \to \alpha$. As a formal algorithm in base form this is

```
definef subst (gram,p); /* replaces the production p: r → ℓ1...ℓn
                                          by 'expanding' ℓ1      */
gram = gram less p + {<ℓhs p, g + rhs(2:)>,g∈ gram{(rhs p is rhs)(1)}};
end subst;
/* and now the algorithm proper.first number the elements of intsymbs, */
/* in some arbitrary order                                    */
place = nℓ; (∀x ∈ intsymbs) place (x) = # place + 1;;
(while ∃p ∈ gram | place((rhs p)(1)) <place (ℓhs p)) subst(gram,p);;
(while ∃p ∈ gram |(rhs p) (1) eq (ℓhs p is x))
     xprime = newat;
     gram {x} = gram {x} - ({v ∈ gram {x}, v(1) eq x} is deleted)
                      + {v + <xprime> v ∈ gram {x}, v(1) ne x};
     gram {xprime} = {v(2:) + <xprime>, v ∈ deleted}+ {v(2:),v∈deleted};
end while;
(while ∃p ∈ gram | (rhs p)(1) not ∈ termsymbs) subst(gram,p);;
```

This algorithm is optimized by conducting the search over *gram* implied by the three preceeding while loops in an ordered manner: for the first two loops upward according to place(<u>ℓhs</u> p); for the last loop, downward according to place(<u>ℓhs</u> p). These improvements amount essentially to three applications of formal differentiation.

(d) Decomposition of a program graph into intervals. The program graph is defined by a set *nodes*, an entry node *exit*, and a node-to-node map *cesor*.

```
definef interval(nodes,x);  /* calculates the interval with head x */
int  = <x>;
(while ∃y ∈ range (int) | {z ∈ (nodes-int) | y ∈ cesor{z}} eq nℓ)
     int = int + <y>;
end while;
return int;
end interval;
definef intervals(nodes, ent); /* ent is the program graph entry node */
ints = {interval (nodes,ent)};
(while ∃nd ∈ cesor [[+: int ∈ ints]range(int) is intnds]-intnds)
                                        interval(nodes,  nd) in ints;;
return ints;
end intervals;
```

More efficient versions of this algorithm can be derived by formal differentiation and procedure integration.

(e) Ford-Johnson Tournament Sort. An informal explanation of this algorithm can be found in O.P.II, p. 66-67, with a SETL representation of it, unfortunately not in base form. A base form of the algorithm is as follows:

```
/* we are given a set of items to sort according to a transitive */
/* binary relation le.To exclude duplicates  we suppose  that the */
/* n  elements of items are pairs with integer second components, */
/* all of which are distinct, and that le is determined from the */
/* first component of a pair only.                               */
definef fordj (items);
if #items eq 1 then return <items>;;
itemcopy = if (# items//2) eq 0 then items else items with
    (<hd ∋ items, [max: item  ∈ items,] item(2)+1>) is newcopy
     /* this forces itemcopy to have an even number of elements */
```

```
map = nℓ;
(while itemcopy ne nℓ)
     x from itemcopy;  y from itemcopy;
     if ℓe (y,c) then map(x) = y else map(y) = x;;
end while;
halfsorted = fordj (hd [map])  /* recursively sort half the elements */
allsorted  = <map(halfsorted(1)), halfsorted(1)>;
                                          halfsorted = halfsorted(2:);
pow2 = 2;
(while halfsorted ne nult)  /* until all elements digested */
     pow2 = 2 * pow2;        /* double length for insertion */
          allsorted = allsorted + halfsorted(1:(pow2 - # allsorted)
                                          min # halfsorted is ntaken);
          (ntaken ≥ ∀n ≥ 1)
                    mergein(allsorted, pow2 - 1, map(halfsorted(n)));
          end ∀n;
          halfsorted = halfsorted(ntaken + 1:);
          /* remove elements that have been passed to allsorted*/
end while;
return if (# item//2) eq 0 then allsorted else
     /* drop added element */ [+: elt(n) ∈ allsorted elt(2) ne newcomp]
                                                                   <elt>;
end fordj;
define margein (vect,lem, elt);
/* this routine takes a sorted vector vect and merges the element elt */
/* into its proper position among the first lim elements of vect. */
/* after optimization,this would be a 'binary search' based insertion */
must = 1 ≤ ∃k ≤ lim | if ℓe (elt,v(1)) then k eq 1 else if
                         ℓe(v(lim), elt) the k eq lim  else
                          ℓe(elt, v(k)) and ℓe (v(k - 1), elt);
vect = vect(1: k - 1) +
     if ℓe (elt, vect(k)) then <elt, vect(k)> else <vect(k), elt>
     + vect (k + 1:);
return;
end margein;
```

As can be seen, this is a construction of nontrivial mathematical complexity.

2. A Comment on Correctness Proofs.

When an algorithm BF in base form is redeveloped in some more highly optimized form, additional code details will appear. Since the optimizations applied to BF will generally be of some relatively stereotyped form, the details which appear after optimization will themselves tend to be stereotyped. For this reason, it will generally be the case that, whereas the 'Floyd assertions' needed to prove the correctness of BF can only be derived by a relatively deep analysis of algorithm semantics, the additional assertions needed to prove the correctness of the optimized form of BF can be set up in a stereotyped way, given that the optimizations applied to BF are known. We may also hope to prove general lemmas characterising the manner in which the optimising transformations applied to BF transform the assertions associated with BF's original form. Because of the importance of these possibilities, it seems reasonable to assert that proofs of algorithm correctness should begin with the base form of an algorithm rather than with any more highly detailed algorithm form.

Examination of a few of the algorithms considered above will confirm this view.

(a) Bubble Sort. It is apparent that if the bubble sort terminates then the vector v will satisfy $v(n) \leq v(n+1)$ for all n. The fact that the components of v are not being changed can be expressed by stating that the algorithm leaves every set $S_a = \{n, v(n) \underline{eq} a\}$ invariant. This fact follows from the fact that each of the elementary 'swap' operations applied to v has this property.

As previously noted, the bubble-sort algorithm can be optimized by turning the existential into a search loop, and by noting that the range of indices in which no ill-ordered pair exists increases when a properly ordered pair is examined, and decreases by at most one when a swap is performed.

The optimized algorithm is then the familar

```
n = 1;
(while n lt # v)
     if v(n) < v(n+1) then
          <v(n), v(n+1)> = <v(n+1), v(n)>;
          n = if n eq 1 then 2 else n-1;
     else
          n = n + 1;
     end if;
end while;
```

(b) Heap Sort. It is evident from the form of the *makeheap* subroutine that on return from *makeheap* the assertion $1 < \forall m \le n \mid v(m/2) \ge v(m)$ must hold. From this it follows mathematically that $1 \le \forall m \le n \mid v(1) \ge v(m)$. Moreover, makeheap (v,n) leaves invariant all components of v with indices larger than n. Hence the following assertions can be added to the main loop of *heapsort*:

```
(# v ≥ ∀n > 1)
     assert n ≤ ∀k ≤ #v, 1 ≤ ∀j ≤ n | v(j) ≤ v(k)
     and    n ≤ ∀k < #v | v(k) ≤ v(k+1);
     makeheap (v,n);  <v(n), v(1)> = <v(1), v(n)>;
end ∀;
```

The assertion holds initially because it is vacuous; at the end of the loop, we have $1 \le \forall m \le n \mid v(n) \ge v(m)$ and $n \le \forall m \le \#v \mid v(n) \le v(m)$, confirming the assertion on the loop path. It follows that on exit from the loop we have $1 < \forall k < \#v \mid v(k) \le v(k+1)$ and $1 < \forall k \le \#v \mid v(1) \le v(k)$, so that on loop exit v is sorted. The fact that during sorting the components of v are not changed can be shown as in the bubble sort case.

(c) Greibach normal form of a grammar. Let L(gram) be the language generated by *gram*. Then L(gram) = L(subst(gram,p)) is one of the two fundamental mathematical facts on which the correctness of algorithm (c) rests. This assertion concerning the two infinite sets L(gram) and L(subst(gram,p)) is not really decomposable into more elementary facts at the algorithm-theoretic level, since the body of the *subst* procedure is simply one single set-theoretical assignment. The equality of these two sets is rather to be regarded as a directly mathematical fact. The fact that transformation of *gram* by

```
xprime = newat;
gram {p(1)} = gram {p(1)} - {v ∈ gram{p(1)}|v(1) eq p(1)}}
      + {v + <xprime>, v ∈ gram{p(1)}, v(1) ne p(1)};
gram {xprime} = {v(2:) + <xprime>, v ∈ gram{p(1)} |
                                          v(1) eq p(1)},
```

as in the second *while* loop of algorithm (c), is in much the same sense primitive and set-theoretic. Given these facts, the fact that algorithm (c) does not change L(gram) is clear. It is also clear that the transformations of *gram* effected by algorithm (c) never enlarge the set s = {(rhs p)(1) p ∈ gram}. Thus, if we set termsymbs = s - intsymbs at the beginning of the algorithm, it is clear from the form of the final *while* loop of algorithm (c) that {(rhs p)(1), p ∈ gram} ⊆ termsymbs at the end of the algorithm.

(d) Decomposition of a program graph into intervals. It is clear from the form of the *while* loop in the *intervals* function that on return from *intervals* that the set s = [+: int ∈ ints] range(int) satisfies cesor[s] ⊆ s. Moreover, s ⊇ range(interval(nodes,ent)). The *while* loop in the routine *interval* clearly never diminishes the set range(int); and since this set is initially {x}, it follows that x ∈ interval (nodes,x) always holds. Thus we can conclude that ent ∈ s. Since it is always assumed of program graphs that the transitive closure of *ent* under *cesor* includes all *nodes*, we may deduce that [+: int ∈ ints] range(int) = nodes.

The assertion

assert $\forall x(n) \in$ int|n eq 1 or (x $\in$ cesor{z} implies
1 < $\exists$m < n | z = int(m));

can be inserted immediately following the *while* statement of the *interval* procedure. Indeed, it holds on the first iteration of the *while* loop since *int* is of length (1); and in virtue of the *while* condition it is clearly preserved during subsequent iteration. Therefore any tuple returned by the *interval* function will only admit forward branches (except to its first component) and can only be entered through its first component. We can show in the same way that each *int* returned by intervals satisfies

(*) **assert** $\forall$x(n) $\in$ int | n eq 1 or (1 $\leq$ $\exists$m < n | x $\in$ cesor {int(m)})

It follows mathematically that if int_1 = interval(nodes, x_1) and int_2 = interval(nodes, x_2), then range(int_1)and range(int_2) are disjoint unless $x_1 \in$ range(int_2)or $x_2 \in$ range(int_1). We may therefore attach the assertions

(α) $\forall int_1 \in$ ints, $int_2 \in$ ints|int_1 eq int_2 or
range(int_1) * range(int_2)) eq nl

(β) $\forall$x $\in$ [+: int $\in$ ints] range(int)|x eq ent or
$\exists$y $\in$ [+: int $\in$ints] range(int)|x $\in$ cesor {y})

to the *while* loop of the *interval* routine. These assertions hold by initialisation when the loop is first entered. The second assertion is preserved by (*) since the first component of each *int* added to *ints* belongs to cesor[[+: x $\in$ ints] range(x)]. It is clear that for each *int* added to ints we have $\forall$x $\in$ ints | int(1) not $\in$ range(x); moreover, by (*) and (β), y $\in$ (range(int) * [+: x $\in$ ints] range(x)) implies y eq ent. If as is customary we assume that *ent* has no predecessor in the program graph, it follows that the assertion (β) is true for all iterations of the *while* loop of the *intervals* routine.

(e) <u>Ford-Johnson Sort.</u> Because of the assumption that the elements of *items* are integers it follows that $\langle$<u>hd</u> $\ni$items, [<u>max</u>: item $\in$ items] item(2) + 1$\rangle$ does not belong to *items*, so that *itemcopy* must have an even number of elements. Thus, after the execution of the first *while*-loop of the *fordj* algorithm, we will have

(α) (domain (map) + range (map)) <u>eq</u> itemcopy,

and also

(β) $\forall x \in$ domain(map) | ℓe (map(x), x).

Moreover, *map* is easily seen to be 1 - 1.
Taking

(γ) (v <u>eq</u> fordj (items)) <u>implies</u> range(v) <u>eq</u> items <u>and</u>
1 $\leq \forall$j < # v | ℓe (v(n), v(n+1))

an an inductive assertion, it follows that after the first recursive call to *fordj* we have

(δ) range(halfsorted) <u>eq</u> domain(map) <u>and</u>

1 $\leq \forall$j < # halfsorted | ℓe (halfsorted(n),halfsorted(n+1)).

As assertions for the *while* loop which follows we use

(ϵ) itemcopy <u>eq</u> (range(allsorted) + range(halfsorted) +
map(range(halfsorted)));

also
(ϕ) (1 $\leq \forall$n < # allsorted) ℓe (allsorted(n), allsorted(n+1)),

and the assertions

(η) $\forall x \in$ range (allsorted) | ℓe (x, halfsorted(1))

immediately before the statement just preceeding the *while* loop, and

(η') ntaken <u>ℓt</u> # halfsorted <u>implies</u> $\forall x \in$ range(allsorted) |
ℓe (x, halfsorted(ntaken + 1))

immediately after this statement; we also use assertion (β) and the second clause of assertion (δ) within this loop. Assertion (β) is invariant, and the second clause of (δ) will continue to hold since *halfsorted* is only being diminished. It is clear that (η´) follows from (η), and (η) from (η´) on the next iteration.

Within the ∀n-iteration imbedded in this *while* loop we continue to make use of assertions (ϕ), (η) and (β), but replace (∈) by

(κ) **itemcopy eq** (range (allsorted) + range (halfsorted(ntaken + 1:))
+ **map** (range(halfsorted(n:)))
+ **range**(allsorted(# allsorted-ntaken + n:));

and

(λ) (#{x(k) ∈ allsorted | not le (map(halfsorted(n)),x)}) ≤ pow2-1.

Note that one statement after the point at which we exit from the ∀n-iteration, assertion (κ) reduces to (∈). On entry to the iteration (κ) follows from (∈), since allsorted = allsorted + halfsorted (1: ntaken) will just have been executed. Only the last statement of the *mergein* routine changes its *vect* argument, and from the form of this statement it is evident that on exit range(vect) has become the union of the entry value of range(vect), plus {elt}, i.e., *allsorted* has become allsorted + {map(halfsorted(n))}. Thus (κ) holds during every cycle of the ∀n-iteration. For the same reason, (η´) holds during every cycle. It is clear from the statement immediately preceeding the ∀n-iterator that on entry to the ∀n-iteration # allsorted is pow2, and that range(allsorted) includes halfsorted(ntaken). Thus assertion (λ) holds on entry to the ∀-iteration. Each time we iterate at most one element is added to range(allsorted), but for any given value of n at least ntaken-n+1 elements of range(allsorted), namely range(halfsorted(n: ntaken-n+1)), belong to the component of the set appearing in assertion (λ).. Hence (λ) holds throughout the ∀n-iteration.

By (ϕ) the *vect* argument to *mergein* is in sorted order when *mergein* is called. By (λ), on each subsequent iteration every component x of *allsorted* such that not ℓe(map (halfsorted(n)),x) has an index less than *pow2*. Hence the k found by the existential in the first statement of *mergein* will satisfy ℓe(elt, v(λ + 1)) each time *mergein* is called (with v = allsorted and elt = map(halfsorted(n)); and thus since only the second statement of *mergein* modifies its first argument v = allsorted, it is apparent that this argument remains sorted after return from *mergein*. We can now conclude that (ϕ) remains true for all iterations of the $\forall$n-loop.

Within the *while*-loop WL containing the $\forall$n-loop, assertion (ϵ) can be seen from (α), (δ), and the way in which *allsorted* and *halfsorted* are initialised just before the loop to hold on initial loop entry; since (κ) reduces to (ϵ) on exit from the $\forall$n-loop (ϵ) holds during every iteration of WL. Assertion (ϕ) holds by initialisation on entry to WL. Using (η') and the second clause of (δ), it follows that (ϕ) holds immediately after *allsorted* is modified by the second statement of WL. Since we have seen that the $\forall$n-loop preserves (ϕ), it follows that (ϕ) holds throughout WL. It is also clear that (β) and the second clause of (δ) hold throughout WL, since *map* is not modified and *halfsorted* is only decreased.

We may therefore conclude that (ϕ) holds on exit from WL; note also that on exit from WL halfsorted eq nult, so that (ϵ) reduces to itemcopy eq range (allsorted). Because of (ϕ) it is clear that v = [+: elt(n) $\in$ allsorted | elt(2) ne newcomp] satisfies

(ϕ^{*}) $\qquad\qquad$ ($1 \leq \forall n < \#v$) ℓe (v(n), v(n+1)),

Moreover, it is clear that

itemcopy eq if (# items //2) eq 0 then items else items + {newcomp}

and that

range(allsorted) = range(v) + if (# items //2) eq 0 then nl else {newcomp} .

We have already noted that newcomp not $\in$ items. Thus in all cases range(v) eq itemcopy; so that inductive assertion (δ) is verified.

3. Debugging of Proofs.

To develop a verified correctness proof for an algorithm in base form we will ordinarily have to

(a) Attach assertions to the formal text T of the algorithm, and verify a number of relationships which tie these assertions to the statements of T and which have the form if A_j can be asserted at point P_j of a program PR, then A can be asserted at point P'. The verification of propositions of this sort will generally be rather routine, as the relationships being verified are all recursive. All that will ordinarily be involved is symbolic manipulation of a conventional sort, which must however be guided by a semantic knowledge of the programming language which is employed.

(b) Step (a) will yield a family of set-theoretic propositions $\bar{A}$, one such proposition being attached to each (significant) point in PR. These propositions $\bar{A}$ will have the form 'if A (which by step (a) is a consequence of assertions attached to other points of PR) holds, then B (a Floyd production directly attached to this point of PR) is implied'. The propositions $\bar{A}$ are of standard set-theoretic form, i.e., are mathematical objects no longer having any explicit tie to the details of PR's statements. To verify PR one must then prove all of the propositions $\bar{A}$. The proof which here becomes necessary should itself be checked by an automatic proof-checker (if indeed it is not constructed by an automatic theorem prover), since merely manual proof elaboration (in something like the style illustrated in section 2 above) leaves open the possibility that a crucial (if perhaps marginal) case is being glossed over erroneously.

Of the steps just outlined, it is the last one, construction of a series of proofs in a form acceptable to an automatic proof checker, that is apt to be the most onerous. Two reasons buttress this expectation. In the first place, proof-checker technology is still only weakly developed; proof checkers can take only very small steps themselves, and must therefore be guided in great detail. Moreover, step (a) above will generally transform the assertions P originally attached to a program in such a way as to obscure any intuitive flavor which these assertions might originally have possessed.

For this reason, it will be quite important in developing proofs of program correctness to ensure that the Floyd assertions with which such a proof begins are in fact correct. However, as originally set up these assertions, which will often be roughly equivalent in bulk to the programs to which they attach, are just as likely to attract numerous small errors as programs are. Of course, this objection falls away for any set of assertions to which we are ultimately able to give a mechanically verified proof. However, in most cases we will not want even to try generating a formal proof until the set of assertions to be proved has been subjected to a preliminary check for plausibility. Thus we expect informal techniques like those presently used in debugging to be useful in the early stages of development of fully detailed program correctness proofs.

One tempting way of checking a set of Floyd assertions purporting to constitute the beginning of a correctness proof is simply to verify that the assertions do in fact remain true as we run the program PR to which they are attached. Dynamic assertion checking can be quite expensive in the not uncommon case in which the assertion being checked contains one or more quantifiers.

To reduce the cost of dynamic checking in such cases, Earley's 'iterator inversion' method (set theoretic strength reduction, which might also be called 'iterator reduction') can be used.

The preceding considerations suggest that the following project might be useful: Build up a partial proof system within which step (a) of a formal correctness proof system is actually implemented, and which also incorporates facilities allowing programs to which Floyd assertions have been attached to be run and the assertions to be checked at run-time. Where feasible, let the run-time checking mechanism incorporate optimizations of the Earley type. Then use this system to annotate a fairly extensive library of base-form SETL algorithms with debugged Floyd assertions. An effort of this kind would throw a good deal of light on the formal proof-checking task which it left as a residue.