SETL Newsletter # 210

Remarks on Debugging

J. T. Schwartz
February 6, 1979

Debugging always starts with evidence that a program error has occured somewhere in the history of a run. The problem in debugging is to work one's way back from the visible sympton to this program error. What one seeks can be called the *error sources* or *primal anomalies*, which are those wrongly stated operations or tests whose immediate consequence is the transformation of a collection of reasonable inputs into an output which is unreasonable in some regard. Of course, the history of an extensive computation constitutes a vast mass of data, impossible to survey comprehensively. The debugging process therefore aims at the exploration of as narrow a path as possible, with the aim of finding one's way back to one or more primal anomalies.

Here it is interesting to compare the two quite different processes of syntactic and semantic debugging. Even if we assume that raw program text (carefully desk-checked but never compiled) may contain as many as 1/10 syntax error per line on the average, the syntactic debugging of a thousand line program normally proceeds routinely and rapidly. The tool that allows this is a compiler with fairly good syntactic debugging aids, among which the following are particularly desirable:

(a) Unambiguous, easy-to-comprehend error messages;
(b) Suppression of spurious error messages generated by prior errors;
(c) A diagnostic capability which does not decay during the parsing of a lengthy, error-rich text.

These capabilities lie well within the present state of the art of parsing. If a compiler with these capabilities is available, the normal syntactic history of a 1000-line text initially containing 100 errors would normally be something like the following:

Compilation 1: 125 error messages generated, of which 75 are genuine; 75 errors corrected, of which 10 are corrected wrongly.

Compilation 2: 70 error messages of which 30 are genuine; 30 errors corrected, of which 5 are wrongly corrected.

Compilation 3: 20 error messages of which 7 are valid; 7 errors corrected, of which 1 is corrected wrongly.

Compilation 4: 10 error messages, of which 4 are genuine. All 4 errors successfully corrected.

Compilation 5: No errors.

In a system providing rapid turn-around, this need not be more than the work of a day or two. Note the important role played by the ability to uncover multiple faults during a single run.

Next consider the process of semantic (i.e.'logical' or 'execution') debugging of the same program. Here we make the more favorable assumption that, owing to careful desk checking and to the elimination of some logical errors during syntax checking, only 50 errors are present in the original 1000-line program. Now the typical iteration is approximately as follows:

(a) The program runs and bombs. Assuming that a miscellany of print statements was included for debugging purposes, the programmer then forms an idea of what has happened (e.g. certain code never reached, wrong argument values passed to certain procedures, unreasonable values detected for certain variables.)

(b) This evidence, considered, will in favorable cases point the finger of suspicion at certain narrow program sections. However, in unfavorable cases, the available evidence may be

quite ambiguous, and may simply lead the programmer to generate considerably more extensive traces and dumps. It is worth describing three possible points along the spectrum of possibilities typically encountered:

(b.i) Within a region of code described as suspicious, at least one visibly incorrect instruction may be spotted and corrected.

(b.ii) A program region containing the error may be correctly described, but no specific error located. In this case, one more run with denser tracing in the error region may locate the anomaly.

(b.iii) The program region first suspected may in fact contain no error. In this case denser tracing will simply confirm the good behavior of the suspected region, after which reconsideration may lead to suspicion being cast, this time more correctly, on some other region.

The following are reasonably typical sequences of steps to uncovering a logical error:

Step 1: Suspect region R, insert traces.
Step 2: Locate and fix bug.
Step 3: Correct syntax error in step 2. (Bug is now fixed).

Alternatively:

Step 1: Suspect region R, insert traces.
Step 2: Region R ok, suspect region R', insert new traces.
Step 3: Correct syntax error in step 2, obtain new traces.
Step 4: Locate and fix bug from new traces.

Overall it can be hard to fix more than 1/3 bug per run, as compared to the estimated average of 25 bugs fixed per run in our hypothetical account of syntactic debugging. Thus 150 runs, which might represent as many as 30 days work, can be required to fix the 50 logical bugs which might very typically be present in a new, 1000-line program.

To alleviate this vexing but all-too fimiliar situation, we must aim to increase the probability of finding at least one logic bug per run(if any is present) and, still better, must make it possible to find more than one bug per run. The following considerations are directed toward this end.

(I) It is best that programs should not run for long after they have begun to generate erroneous quantities, since the longer they run the more remote the primal anomalies will become. SETL strives for this desideratum in two ways:

(I.A) Erroneous programs will often generate Ω quantities, and attempts to use Ω's will cause blowup.

(I.B) A program being debugged should be thickly larded with ASSERT statements. If this is well done, the probability of a logical error leading to quick blowup should become quite large.

II. Enough information to make it possible to trace back to a primal anomaly should be dumped routinely upon program blowup. What seems desirable here is to dump the last value assigned to each variable X by every statement that modifies X. If this is done, a primal anomaly will only be hidden if the instructions I which constitute it generate an erroneous result R, pass R along as inputs to the instructions which will eventually (and probably soon) develop an error symptom from R, following which I is somehow re-executed, this time producing a correct-looking result which hides the erroneous character of I. Such tricky situations are of course possible, but unlikely.

To generate such a comprensive dump of last values assigned, we can proceed as follows: As a program P runs, a count can be kept of the number of times that each basic block within it is executed. If and when P fails, these counts will be available. The program can then be incremented, and these counts decremented as execution proceeds. Each time a count reaches zero we know

that we are entering a block for the last time. Wherever this happens, we can switch the block into an alternate mode in which variables are printed each time they are modified (along with an indication of the statement which is effecting the modification.) The cost of this is only a doubling of the normal execution time of an erroneous run, which is probably a smaller cost than would be incurred by the less systematic process of ordinary debugging.

On failure it is also appropriate to dump an indication of routines currently called, with the values of their parameters, and of control-flow history. This history can consist of a statement of all branches recently taken, with an indication of number of times taken if a given branch is taken repeatedly in the same way.

III. Next we turn to the question of how to discover more than one bug per run. Here a simple but attractive proposal can be based upon the subprocedure structure of a program, which will normally be a directed graph, though in recursive cases a few simple mutually recursive loops of programs can be present. When this is the case, subprocedures can be debugged by a proceeding up the directed call-graph, from invoked procedures to the procedures which invoke them. Whenever a layer of mutually independent procedures is encountered, they can be debugged in parallel by invoking each of them with plausible test data. To facilitate this, it may be worth using one special syntactic construct, e.g. ERROR.

We suppose that the ERROR quantity (syntactically a Boolean value) can only be used in the main part of a SETL program, and originally has the value 'false'. Then, if the SETL machine detects an error during execution of some subprocedure, it will force return from all aubroutines, and transfer to the last point at which ERROR was evaluated, there resuming execution, and giving ERROR the value 'true'. Using this construct, a debugging plan for a program involving two level 1 procedures and two level two procedures might be as follows:

```
      (here set up parameters  A.1)
      if ERROR  goto  TRYB1;
      PROC_LEV1_A (parameters  A.1)
TRYB1: if  ERROR  stop;
      (here set up parameters B.1)
      PROC_LEV1_B (parameters B.1)
      if ERROR  goto  TRYB2;
      (here set up parameters  A.2)
      PROC_LEV2_A (parameters  A.2)
TRYB2: if  ERROR  stop;
      (here set up parameters  B.2)
      PROC_LEV2_B (parameters  B.2);
```

This could double the number of anomalies found per run, assuming that each subprocedure tested shows at least one anomaly. Of course, error dumps should be generated each time an error is found.

It is worth noting that very high level languages have a real advantage here, since the inherent data structuring which they provide reduces the number of procedures which need to receive elaborate argument values having special, non-standard structures. This should make argument setup very much easier.

Another more sophisticated approach to discovery of more than one anomaly per test run is worth suggesting. Ordinarily, quite a few features of programs are generated by an implicit optimising process of 'set theoretic strength reduction' or 'iteratator inversion' in the manner described by Paige. This optimization introduces variables x which carry the values of expressions $e(y_1,\ldots,y_n)$ that would otherwise have to be calculated repeatedly, but makes it necessary to update the value of x whenever $y_1,\ldots,y_n$ are modified, a requirement that can easily lead to error either because an update operation is forgotten or because it is wrongly expressed. In this situation, there will naturally arise assertions of the form

ASSERT: $x = expn\ (y_1,\ldots,y_n)$.

We can then change the syntax of such assertions to

ASSERT: $x := expn\ (y_1,\ldots,y_n)$,

and agree that assertions having this latter form which fail will generate appropriate dumps but assign *expn* to x and continue execution. In many cases, this will allow defective programs to continue correct execution, up to the point at which one

or more additional anomalies are uncovered. To generate the necessary dumps without increasing execution costs significantly, an execution count technique generalizing that outlined above can be used.