Constructing Quality Software, P.G. Hibbard/S.A. Schuman (eds.)

SETL AS A TOOL FOR GENERATION OF QUALITY SOFTWARE *

Robert B. K. Dewar
Arthur Grand
Ssu-Cheng Liu
Edmond Schonberg
Jacob T. Schwartz

Computer Science Department
Courant Institute of Mathematical Sciences
New York University

'Pure' SETL is a language of very high level, allowing algorithms to be programmed rapidly and concisely with minimum attention to specification of detailed data structures. A representation sublanguage adds a system of declarations which allow the user of the language to specify the data structures that will be used to implement an algorithm which has already been written in pure SETL, without necessitating any rewriting of the algorithm itself.

To the extent that programming tools can aid in the development of quality software, there seem to be two possible approaches to their design:

1) Make it harder to write bad programs

2) Make it easier to write good programs.

Much of the work in development of very strongly typed languages (e.g. ALPHARD, [1]) which emphasize aids for proof of correctness approaches seems to focus on the first aim of making it hard to write bad programs. The difficulty with this approach is that the syntactic complexity imposed on the programmer often conflicts with the goal of making it easier to write good programs. Thus we may observe that even simple programs in ALPHARD are long and involve a large proportion of text which is not part of the algorithm proper, but rather represents redundant semantic information whose sole purpose is to facilitate formal proofs of program correctness.

The approach of SETL [2] is to focus almost exclusively on the aim of making it easier to write good programs, realizing that this involves omitting some of the built-in redundancy which is characteristic of strongly typed languages. It is interesting to note that two other languages taking a similar approach, APL and SNOBOL4, are proving to be extremly popular despite minimal manufacturer support and despite the fact that both these languages can

* This work was supported by: The National Science Foundation Grant MCS-76-0016, and ERDA Adminstration Agency Contract EY-76-C-02-3077*000.

only be dismissed as jokes by proponents of structured programming concepts. We believe that the success of these two languages is a reflection of the fact that the emphasis on ease of programming is an important one. The SETL language attempts to capitalize on this point, and at the same time demonstrate that this goal can be achieved without sacrificing the basic notions of modularity and structure which are essential to the production of large programs.

To illustrate the SETL approach, we give in Fig. 1 a complete example of a SETL program. This program performs a topological sort using the familiar algorithm which keeps track of the number of predecessors of each node in the graph. The input data consists of pairs of strings, representing the ordering relationship. The output is a list of strings giving one possible linear ordering. For the sake of simplicity, a test for cycles has been omitted.

```
module main;

      /* names is set of names involved        */
      /* succ is a map giving the ordering     */

      names  := nl;
      succ   := nl;

      /* loop reading pairs giving ordering    */

      loop doing read(a,b) until a = om do
           names with a;
           names with b;
           succ  with [a,b];
      end;

      /* numpred(x) = number of predecessors of x  */
      /* nopred is list of nodes with numpred = 0  */

      numpred := {[n,0] : n in names};
      (forall [a,b] in succ) numpred(b) + 1; end;
      nopred := {n in names | numpred(n) = 0};
      /* this loop picks elements from nopred and  */
      /* removes them, adjusting numpred properly  */
      (while nopred /= nl)
           print(next from nopred, newline);
           (forall f in succ(next))
                numpred(f) - 1;
                if numpred(f) = 0 then nopred with f; end
           end;
      end;
end;
```

FIGURE 1. TOPOLOGICAL SORT IN SETL

This example shows how SETL allows specification of an algorithm of this type with a minimum of extraneous details. No declarations are required, all typing being weak, and it is not even necessary to specify the data structures which will be used for the various maps and sets involved. However, the basic details of the algorithm must be provided. SETL makes no attempt to derive clever algorithms automatically from high level specification statements. To illustrate this point, we give another program for the topological sort in Fig. 2. SETL regards this as an (inefficient) algorithm in its own right. The problem of deriving the algorithm of Fig. 1 automatically from that of Fig. 2 is an interesting one, but is not attempted in the current version of SETL, nor envisioned as an integral part of the language. For an approach to this problem using techniques of formal differentiation, see [3].

```
module main;

      names := nl;
      succ  := nl;

      loop doing read(a,b) until a = om do
           names with a;
           names with b;
           succ  with [a,b];
      end;
      (while is n in names | not(is x in names | [x,n] in succ))
           print(n from names, newline);
      end;
end;
```

FIGURE 2. SLOW TOPOLOGICAL SORT IN SETL

Returning to Fig. 1, this program may be submitted to the SETL compiler and will run as it stands, using a standard hash table structure for all maps and sets. The execution will be efficient in the sense that the algorithm is linear in the number of pairs in succ, and its execution time will be linear when expressed in this manner in SETL, i.e., no hidden 'order N' or worse inefficiencies creep in. However, the running time will be worse (by some constant factor) than the low level expression of the same algorithm due to the basic overhead of accessing hashed structures.

One possible approach to improving the efficiency of the algorithm in Fig. 1 is to recode it in some lower level language where explicit data structure choices, which are more efficient for the problem at hand than hash tables, can be made. To illustrate this we show just the last part of the algorithm (the loop which outputs successive elements) coded in ALGOL-68 with two different data structure choices:

Fig. 3 shows the result of using linked lists to represent the map *succ* and the set *nopred*. This data structure choice ensures linear time behaviour of the algorithm.

Fig. 4 shows the result of using a bit matrix to represent the map *succ*, and a bit string to represent the set *nopred*. This representation is more efficient in terms of storage use, at the expense of added execution time and a departure from linear behaviour.

```
      mode node = struct (string name,
                          int    numpred,
                          ref link succ);
      mode link = struct  (ref node node,
                          ref link link);
      ref link nopred;
comment (while nopred /= nil) comment
      while nopred :/=: ref link (nil) do
comment print(next from nopred, newline); comment
      ref node next = node of nopred;
      nopred := link of nopred;
      print ((string of next, newline));
comment (forall f in succ[next]) comment
      ref node fn := succ of next;
      while fn :/=: ref link (nil) do
           ref node f = node of fn;
           fn := next of fn;
comment numpred(f)-1; comment
           numpred of f -:= 1;
comment if numpred(f) = 0 then nopred with f; end; comment
      if numpred of f = 0
      then
           nopred := heap link := (f, nopred)
      fi
    od
 od
```

FIGURE 3. ALGOL-68 TRANSLATION (LINKED LISTS)

```
      int n := # number of items involved #;
      [n,n] bool succ;
      [n] int numpred;
      [n] string name;
      [n] bool nopred;
comment (while nopred /= nil) comment
```

```
      while
      int next;
      bool found := false;
      for j to n while not found do
           if nopred(j) then found := true fi
      od;
      found
      do;
comment print(next from nopred, newline); comment
      nopred[next] := false;
      print((name [next], newline));
comment (forall f in succ[next]) comment
      for f to n do
           if succ[next, f] then
comment numpred(f)-1; comment
           numpred[f] -:= 1;
comment if numpred(f)=0 then nopred with f; end; comment
      if numpred[f]=0 then nopred[f] := true fi;
    fi
  od
od
```

FIGURE 4. ALGOL-68 TRANSLATION (BIT STRINGS)

From these examples we note two important points. First, the expression of the algorithm in ALGOL-68 is greatly affected by the choice of data structures. It is quite difficult to see, from the ALGOL-68 text alone, that the underlying algorithm is identical since its logical structure has been obscured by different data structure choices in the two cases.

Secondly, the derivation of these examples from the basic algorithm as expressed in SETL is essentially mechanical once the data structure choice has been made. To emphasize this point, we have included as comments in the ALGOL-68 text the SETL statements to which sections of the ALGOL-68 text correspond.

The fact that this translation is mechanical suggests that it should be automated, and this is the central idea behind the SETL representation sublanguage, whose purpose is to allow the programmer to gain the efficiency made possible by detailed specification of data structures, without requiring the time consuming and error prone process of translating the entire algorithm.

The declarations of the representation sublanguage are completely separate from the SETL program itself, which is not modified in any way. The normal procedure is to create these declarations as a separate file, which the compiler merges with the original program by matching up module and procedure names.

At its simplest level, the representation language allows datatype declarations which are syntactically similar to the ordinary declarations of an ALGOL like language. The intention is however focused on gaining efficiency, rather than obtaining the compile time checking available with strong typing. The resulting program will run faster simply because some type checks may be eliminated, and hard code generated for some sequences which required library calls in the original form. However, the program still contains a sufficient number of type checks to ensure its integrity. If there is an assignment statement A: = B; and A has been declared to be an integer, and B has been left undeclared, then if it is not possible to determine by global flow tracing that B has an integer value at the point of the assignment, a type check is made and an execution error occurs if B is not an integer. This ensures that the only possible adverse effect of making such datatype declarations is to cause the resulting program to terminate with an error message. It is never possible to introduce a type error which is undetected and causes erroneous results.

The more complex part of the representation sublanguage is concerned with specifying the data structures for maps and sets. The important concept here is that of a base set. A base set is an auxiliary set which may not appear explicitly as a variable in the program, and in terms of which actual program variables are described.

In the case of the topological sort, the significant base set is the set of names, and a statement:

```
base nodes : string;
```

identifies the base and gives it the name nodes. This name does not appear in the algorithm itself, but will be referenced in other declarations in the representation section.

We then identify the program variable *names* as a set whose elements are elements of this base set:

```
repr names : local set {elmt nodes};
```

The base set will be stored as a linked hash table at run time. The repr declaration for *names* causes it to be represented using a single bit associated with each element of the base set, the bit being on if the element of the base is in names. In this particular case, all such bits will be on, since the base nodes is identical to the set *names*. However, this is a property of the algorithm on which we do not depend when constructing these declarations.

Two other representation are available for sets, and are illustrated by the following possible declarations for the set *nopred*:

```
repr nopred : sparse set {elmt nodes};
repr nopred : remote set {elmt nodes};
```

The sparse representation causes *nopred* to be stored as a hashed linked list of pointers into the base set. Normally SETL does not use pointers to values (an assignment of a set or map value causes the map to be copied, rather than generating a pointer to the same value), however this introduction of pointers is entirely transparent, since the base set is not a program variable. Thus every

time a value is added to the set *nopred*, it will be added to the base set nodes if it is not there already, but this side effect is transparent because the value of the base set nodes cannot be inspected by the program. Again it is a property of the algorithm, not used in the construction of the declarations, that whenever a value is added to *nopred*, it is already in the base set. The representation obtained for *nopred* by this declaration is similar to that chosen in the linked lists ALGOL-68 version of the algorithm.

The remote representation causes *nopred* to be stored as a bit string. As a base set is constructed, unique (but arbitrary) integer index values are assigned to its elements. These index values are used to select the bit corresponding to a particular base element from the bit string. This representation corresponds to that chosen for the second ALGOL-68 version. It should be noted that this representation is particularly effective if set union or intersection operations are to be performed on sets stored in this manner, since these operations reduce to bit vector logical operations.

We now turn our attention to the maps *numpred* and *succ*. There are three representations available for maps where the domain of the map corresponds to a base set, analogous to the three representations for sets:

local	map	Represented by storing the range value in a dedicated field allocated in each element of the base set.
remote	map	Represented by storing a vector of range values, indexed by the index values associated with base elements.
sparse	map	Represented by a separate hashed linked list, where the domain value of each entry is a pointer to the corresponding base element.

In addition, the representation declaration distinguish between single valued maps (smap) and multiple valued maps (mmap).

We may now choose representations for the maps *numpred* and *succ* as follows:

```
repr numpred : local smap (elmt nodes) int;
repr succ : local mmap (elmt nodes) sparse set (elmt nodes);
repr succ : remote mmap (elmt nodes) remote set (elmt nodes);
```

The two declarations for *succ* correspond to the two possible choices illustrated by the ALGOL-68 programs. As can be seen, the difference is obtained by altering a single word in the declaration text; no changes whatever are made to the original SETL algorithm, in contrast with the extensive differences in the ALGOL-68 programs. Finally, we have

```
repr n, a, b, f : elmt nodes;
```

This causes these variables to be stored as pointers to the corresponding base elements. Indexing a local map with such a variable corresponds to picking up the appropriate field of the base element using this pointer, which is analogous to a structure selection in

ALGOL-68. Indexing a remote map with such a variable is a vector indexing operation.

The two sets of declarations, corresponding to the two ALGOL-68 examples, are given in their entirely, as they would be submitted to the SETL compiler, in figure 5.

```
Module main;
      /* repr section for list representations */
      base
           nodes : string;
      end;
      repr
           names  : local set {elmt nodes},
           succ   : local mmap {elmt nodes} sparse set {elmt nodes},
           numpred: local map (elmt nodes) int,
           nopred : sparse set {elmt nodes},
           n,a,b,f: elmt nodes;
      end;
end;

Module main;
      /* repr section for bit string representation */
      base
           nodes : string;
      end;
      repr
           names  : local set {elmt nodes},
           succ   : remote mmap {elmt nodes} remote set {elmt nodes},
           numpred: local map (elmt nodes) int,
           nopred : remote set {elmt nodes},
           n,a,b,f: elmt nodes;
      end;
end;
```

FIGURE 5. SETL REPR SECTIONS FOR TOPOLOGICAL SORT

By submitting the original SETL program together with one of the two possible repr sections to the SETL compiler, a resulting program is obtained which closely resembles the corresponding ALGOL-68 program in execution style and efficiency.

The word 'closely' in the above paragraph is worth examining in more detail since it illustrates at the same time the power of the SETL approach and the danger of manual translations such as those given in ALGOL-68 earlier.

The bit string representation in SETL is a very close match for its ALGOL-68 counterpart. The only difference is that the SETL structure for *succ* is a vector of vectors, rather than the two dimensional array which will be obtained from most ALGOL-68 compilers (i.e., it corresponds to declaring *succ* as [] ref [] bool rather than [,] bool).

The 'linked list' representation in SETL is a different matter. SETL will store *nopred* as a hashed linked list, the hashing being required by the fact that *nopred* is a set and sets may not contain duplicate elements. This means that the statement:

```
nopred with f;
```

involves a hashing operation, even in the presence of all the repr statements, causing the program to run more slowly. What went wrong?

The answer is that the ALGOL-68 program is not an exact translation of the SETL algorithm as given. It translates the offending statement without any attempt to eliminate possible duplicate elements. However, it is a fairly subtle property of the algorithm as stated that there never arises a case in which an attempt to insert duplicate elements is made. Thus the ALGOL-68 translator was either clever or lucky, but was in any case doing much more than translating the original algorithm. This is good example of the kind of potential error which can be introduced by manual refinement of algorithms. The correctness of the translation depending on an unstated assumption whose validity, even once stated, is far from obvious. Many program bugs are introduced in similar fashion by basing program transformations on similar assumptions which are not valid in all cases.

The SETL approach guarantees that the program resulting from use of the repr statements is semantically equivalent to the original. In the absence of datatype errors, which will result in error termination, the results obtained will be identical to those obtained from the original program. Therefore it was impossible to use repr statements to convert the SETL program into the (potentially different) version represented by the ALGOL-68 program. It is possible that a clever optimizer could note that such a translation was possible, but this is outside the scope of the current work.

In SETL terms, the approach embodied in the ALGOL-68 linked list translation represents a (slightly) different algorithm, and this difference must be reflected in the original statement. If we use a tuple rather than a set for *nopred*, then the desired semantics (of ignoring equal elements) are obtained. This is achieved simply by changing one statement in the original program:

```
nopred    := {n in names | numpred(n) = 0};

    becomes

nopred    := tuple [n in names | numpred(n) = 0];
```

Repr statements can now be given to generate the linked lists of the ALGOL-68 example essentially exactly. However, the bit string representation for *nopred* is no longer valid and the second ALGOL-68 translation is now "incorrect". Therefore it is no longer possible to generate this translation using repr statements.

although the map *succ* could still be represented as a bit matrix.

In conclusion, the SETL approach allows algorithms to be written in a maximally concise manner, without concern for selecting the data structures to be used. A reasonably efficient execution can be obtained from this initial statement. If greater efficiency is desired, then repr statements may be added to the SETL program at selected points to enhance its efficiency without any danger of introducing errors by subtle departures from the semantics of the original program. The resulting program will execute with efficiency comparable to that attainable from a lower level language without the need to entirely recode the algorithm at the lower level. We believe that this approach will be effective in meeting the declared goal of making it easier to write good programs.

REFERENCES

[1] Wulf, W.A., R.L. London & M. Shaw, *Abstraction and verification in Alphard*: in New Directions in Algorithmic Languages 1975 S.A. Schuman (ed.) IRIA, Rocquencourt, 1976.

[2] Schwartz, J.T., *On Programming: An Interim Report on the SETL Project: Part I: Generalities: Part II: The SETL Language and Examples of its Use:* Revised June 1975 Computer Science Department, Courant Institute Of Mathematical Sciences, New York University.

[3] Paige, R. and Schwartz, J.T., *Expression Continuity and the Formal Differentiation of Algorithms*: Proc. Fourth ACM Symposium on Principles of Programming Language, Jan. 1977.

Holager: It seems to me that sets in the mathematical sense are too general for use in programming. What I think we need is a limited form of sets, in order to get more discipline in the programming process. Have you any comments on that?

Dewar: To some extent, I am put in the position of someone defending APL, trying to argue that the whole world consists of vectors of varying length. I think I have a more secure basis in claiming that all the world is made up of sets and maps, but seriously, I think the test is whether it works in practice. It's certainly not the case that maps are an immediately natural way of thinking, but our experience is that, with some small level of exposure, it quickly becomes a way of thinking which effectively meets the criterion of producing programs in a clear form. I think that it is very hard to demonstrate by any abstract arguments. I'd rather demonstrate it by concrete experience.

Holager: My point was that a set of some limited kind of elements might be a better concept than just a general set of anything.

Dewar: That comment seems related to the question of strong typing in general. There is a very simple answer to that. How can you ever have a set of specific elements without having to say what kind of elements they are? It is a nuisance which does not achieve anything from the point of view of making it easier to write good programs. It may achieve something from the point of view of making it hard to write bad programs, but as I have said, this is not the focus of our objectives.

Wichmann: I don't quite understand how you can change the representation of a procedure parameter, since the representation of the actual parameter may change from call to call. Could you explain please?

Dewar: Each variable that appears in the program has some specific declaration. If there arises a case where you want to give multiple declarations of something, there is a facility for obtaining multiple representations throughout, but there is no facility for saying dynamically represent it sometimes one way and sometimes another way. In most programs a given variable has one use throughout the program. Strong typing does not severely restrict the power of expression. It is, after all, not the case that all variables in an ALGOL 68 program are unions. Although you can construct examples where the latter possibility is wanted, looking at the considerable library of programs which we have collected, that's certainly not a problem.

Koster: If you choose a different representation for your underlying data structures, then you have to change the ALGOL 68 program radically. However, I consider the fact that this program is not invariant under the choices of

representation to be a programming error in the original program. The program, in fact, does not have a sufficient degree of uniformity of referents. If you want to change your representation in this way, you have to separate your program out into a functional part and a representational part, and take care that the changes in the representation are all confined to the layer where this representation is defined.

Ross: Could you compare the SETL approach to other forms of abstract data types, as in CLU or ALPHARD? They all seek to raise the abstraction level. In the early 60's we chose not to extend the syntax in these areas, but only to work with the semantic level in terms of hierarchies of functions. High level functional instructions stayed exactly the same in the face of radical changes of the basic functions which effectively implemented them.

Dewar: First of all, I find it strange to put SETL in the same class as ALPHARD because I find them to be two sides of a coin. We are very interested to know which way that coin will fall. [Ross shows that the coin falls on its side. ☺] I would like to take the example from the ALPHARD paper presented here. I wrote the symbol table from that paper as an exercise. It's about 12 lines of SETL which I considered to be very clear. In fact, the level of clarity is, I think, very similar to, or an improvement on, the level of clarity of the ALPHARD specification section. Unfortunately, there is a lot left to do in ALPHARD after writing the specification section, which is itself twice as long. Now, I am willing to believe that the 150 lines that represent the symbol table example in ALPHARD are correct, after much effort in verifying them. But I am also, with much less effort, willing to believe that my 12 lines are correct. I think we have two very different approaches. I consider ALPHARD to be a crystallization of the approach of doing extra work to avoid errors. The reason that I didn't treat a symbol table example in my talk is because its native implementation in SETL was perfectly efficient and didn't require any use of the data structuring mechanism. The important point in SETL is that these mechanisms would be applied in a very limited manner, only where they were needed to improve efficiency.

Correll: I would come back to the comparison with CLU and ALPHARD. What I like about SETL is that it is easier to write programs in this language; it was very hard for me to write programs in CLU and ALPHARD. The reason is that the approach is different. SETL tries to be a high-level language and tries to support automatic translation to lower levels. CLU and ALPHARD leave this translation process to the user and hope that the user can prove that his implementation is correct.

Dewar: Yes, I certainly agree with that. I might say that in preparing my slides, I found the task of translating into ALGOL 68 more difficult than I expected and, in fact, when I showed the slides to someone who knows ALGOL 68 very well, the one with linked lists and references had to be examined very carefully to make sure that it was correct.

Fuksman: Do you have any experience in writing system programs using SETL?

Dewar: The term "system programming" seems to cover a number
possibilities. Certainly for such system components as compilers, SETL is
appropriate choice. The global optimizer for SETL is being written in SETL a
we expect it to be adequately efficient. Now if systems programs mea
operating systems, I think that raises entirely different issues. For exampl
the question of protection. It is one thing to talk about bad programs; it's qui
another to talk about protection in the presence of malicious syste
components. That problem is not addressed by the current design of t
language.