SETL Newsletter # 117

E. Schonberg
October 4, 1973

A Static Debugging System for LITTLE.

The process of debugging can be roughly divided into two phases:

a) Elimination of lexical and syntactic errors.
b) Elimination of semantic errors.

Phase a) is straightforward in principle, as errors of type a) are invariably detected by the lexical scanner or by the parser. Diagnostic messages are generally explicit enough to make error correction an easy matter.

Phase b) is considerably harder, and constitutes of course the true field of action of the programmer. At the highest level, semantic errors are highly non-local flaws in the design of an algorithm, errors hidden in its global logic e.g., the existence of potentially infinite loops, or the destruction by one part of a program of data-structures needed by some other portion of it. At a lower level but still semantic lie data-type errors more local in nature, i.e. a kind of error independent, or only weakly dependent, on program flow. In contrast with semantic errors of the most global kind, algorithms do exist for the compile- time detection of this latter sort of semantic error. At the simplest level, such algorithms will verify that the arguments of each operator in a program are of a data-type to which the operator legally applies. The data-types and operators to which checks of this sort are usually applied are those explicitly provided by the language; these consistency checks can in fact be handled directly by a parser at the cost of adding a numbers of productions to the grammar of the language, (making these checks explicity syntactic.) In the case of languages where data-types are allowed to vary dynamically (as is the case for SETL) this type of checking becomes more complex, as one has to take program flow into account explicitly in order to follow the effect of successive assignments to the same variable.

In the case of LITTLE, where in principle only one data-type is provided, namely the bit-string, this approach will not yield anything more than what is obtained already from the parser. However, LITTLE'S bit strings are used in a stereotyped way to build the data-structures that the user has in mind when designing his algorithm. The LITTLE field-extractors allow these data structures to be quite elaborate. The realization of these data structures in terms of bit-strings contains implicitly a description of the semantics of the algorithm. Building on this observation, the system to be outlined in this note will allow a LITTLE user to describe, in relatively simple fashion some of the semantic content of his program: The system will then check at compile-time that the instructions in the program are in accord with the semantic description it has been given. As a simple example of what we have in mind, consider the use of the macro eptr in the SETL run-time library. It is useful to specify that this macro represents a field-extractor, and that when applied to a SETL object it extracts a *pointer* out of it. Furthermore, *pointers* can be used as indices to dynamic storage, and in restricted fashion in arithmetic statements, but not as masks or as operands of boolean operations in general. Once this information is supplied to the system we have in mind, the sequence:

```
arg2 = eptr arg1;
temp = param + arg2;
```

will be seen as valid, and the variable temp will be recognized to be a pointer also (provided param is known to be an integer) while the sequence:

```
arg2 = eptr arg1;
temp = param. and. arg2;
```

will be flagged as being objectionable.

The preceeding example indicates that field extractors corresponding to specific named fields should be treated as prefix operators, and that semantic description of both the data-types to which these extractors apply and of the data-types extracted by them is required. We emphasize that the scheme proposed here is totally static, i.e. that no account is taken of flow of control within the program. Each variable will be assumed to retain the same type throughout its scope, or at worst to have one of several types declared for it at the onset of the program.

The implementation of this scheme requires that we make some simple additions to the LITTLE grammar, that we build certain data-structures, and that we design an algorithm which examines the parse tree of a statement (after its validation by the parser) and assigns data-types to expressions and variables. We proceed to discuss each one of these operations in some detail.

A. Type declarations: Additions to the LITTLE Grammar.

We add the following declaration forms to the grammar of LITTLE:

1 - a declare statement of the form:

1.1 declare / vartype1/ var11, var12...var1n / vartype2/var21...

/ vartypek/...;

where vartype1, vartype2, etc. are names freely chosen by the user to designate the type of a variable, e. g. charstring, hashindex, roottuple, etc.; and var11... are names of sized variables. For the scheme to be at all useful, all sized variables must be declared to have some variable-type. We allow variable-types with no variables associated with them, however, as these may be useful in connection with expressions, and also in connection with parameter-passing in procedure calls and returns (specially if those are done through global variables which are not explicitly declared in the program).

Array variables, which are both sized and dimensioned, will appear in a type-declaration statement like the one described above, as having the type array. They will be further described by a statement of the form;

1.2 declare array arrayname (indextype), arrayentry;

indextype and arrayentry are both variable-types.

2. The semantic description of operators (and field extractors) will take a different form. For prefix operators and field extractors, it is natural to consider the declaration:

2.1 from vartype1 get vartype2 using opname;

The user has no possibility of creating new infix operators, but he must specify how the ones present in the language act on his variable-types. The declaration:

2.2 from vartype1 and vartype2 get vartype3 using opname;

will convey this information.

It will be convenient to keep in the system information about the commonest variable-types and operators (integers and arithmetic operators, bit-strings and logical operators) to save the user from the burden of oft-repeated declarations.

3. The declaration statements for functions and subroutines are somewhat complicated by the fact that both can have side effects, and that they may use global variables for linkage. We need to declare the valid data-types of the calling parameters and of the global variables used for input. We also need to declare the assignments made by the procedure to its calling parameters and to global variables that may be affected by it. In the case of functions, we also have to specify the variable type of the value returned explicitly by the function. This information can be conveyed by the following statement-form:

declare procedure procname (typeinl,... typeink),(typeoutl,typeoutn),

((globinl, gtypeinl),....(globinj, gtypeinj)),

(globoutl,gtypeoutl), ...(globoutk,gtypeoutk));

The procedure name is followed by four lists:

a) The list of valid variable-types of the input-parameters to the procedure.

b) The list of variable-types assigned to its output-parameters by the procedure. In the case of a function, this list will have only one element.

c) The list of valid variable-types of global variables used for input.

d) The list of assignments to global variables that describe the side-effects of the procedure.

It should be pointed out that by taking into account side-effects of procedures, the system we describe ceases to be strictly local: after a procedure call, some global variables will be known to have some variable-type, and this information will be used to validate subsequent code.

B. Implementation.

1. Data Structures

The declaration statements just described are used to build tables that collect information on variable-types and operators. These table are the following:

a) A property table *proptab* containing all variable names and their assigned types. In the SETL algorithm that follows we will assume that *proptab* is a set of the form:

proptab = {< var1, { vartype11,...vartype1n } >,···

<vark, { vartypek1,...vartypekm } > };

several types can be assigned to a single variable, which may be used for various purposes in a program.

b) an infix-operator table, *infixtab*, of the form

infixtab = { < infop1, { < arg1ℓ, arg1r, result1 > ,

< argkℓ, argkr, resultk > } >,

··· < infopk, { < >, < > } > };

each entry is a pair whose first element is an operator name; the second element is a set of triples; each triple consists of a pair of valid variable-types for the arguments of the operator, and the variable-type of the result obtained by applying the operator to them.

c) a prefix-operator and field extractor table, of similar form:

prefixtab = { < prefop1, {< arg1, result1 >, <argk, resultk> } >,

··· <prefopj, { < >...} > };

d) an index and array table

e) A table *proctab* describing subroutines and functions. The entries in this table will be almost identical to the body of the declare statements for these procedures. For each of Them there will

be a list of variable-types for the calling parameters; a list of variable types for the output parameters (or in the case of a function, for the value returned by it); and 2 lists of pairs, one for input and one for output global variables. The SETL data-structure assumed is:

```
proctab =  {< procname1, <typein1,...typeink>,<typeout1,...typeoutk>,

        { < globin1, gtypein1>....}, { <globout1,gtypeout1>...}>,

          <proknamek    <  >, <  >, { }, { } > } ;
```

Once these tables are built, the semantic description of the program embodied in them can be checked for completeness. This raises a number of interesting questions which we shall bypass, except for the following simple remark: if we consider the variables in a program as the nodes of a graph, the operators as edges between nodes, and called procedures as links between subgraphs, then the graph representing the complete semantic description given by the user has to be connected. Otherwise, either the semantic description given is incomplete or incorrect, or else the program is actually made up of disjoint pieces which do not interact, and should be divided accordingly.

It might be worth to investigate further the information contained in the connectedness structure of this "semantic graph" of a program.

3. Main algorithm.

Once the tables described in the preceeding section are constructed, a tree-walking procedure can examine the parse-tree of the declared variable types to each node. If the statement is an assignment statement a check against *proptab* is made for the variable on the left-hand side of the assignment. If at some point an invalid operation is detected, the corresponding node is assigned the value *undef*. This value propagates upwards and generates a diagnostic at the statement level.

The main procedure is a function *staticval* which is applied recursively to the nodes of the parse tree. It calls upon specific validating procedures:

validinf validpref validindex, and *validproc*, each of which scans the approprate table, and returns either a variable type or the value *undef*. A global flag *valid* is used to propagate any detected invalid operation upwards rapidly. The validating procedures also output the appropiate diagnostic messages.

In the SETL algorithm that follows, we assume that the parse-tree is represented by a nested tuple whose elements are sub-trees or twigs (corresponding to lexical types). Operators and operands are grouped in the usual inverse-polish ordering, so that for example the statement

```
C = A+B;
```

has the representation:

```
< = C < + A B > >
```

The following forms are assumed for the specific statements noted below

a) indexed retrieval: V(i):

```
< index, V I >
```

b) if statement: `< ifstat, cond, label >`

c) subroutine call: `< call, subname, < arg1... argn>>`

d) A function call with complex parameters will be represented by a similar tree, so that

The statement X = f(y) will have the representation

```
< =, X, < call, f, y > >
```

We also assume that named field extractors are treated as prefix operators, so that their names appear in the parse-tree, instead of their macro-expanded forms.

Finally, we require that the user include in his type declarations all the constants mentioned in the program, so that entries will be made for them into *proptab*. We emphasize again that the entry corresponding to a given variable is not a variable-type but a set of them. This will actually reduce the number of declarations that the user has to make; by assigning a type whose relationship to certain operators is known through other declarations to a variable, we make it unnecessary to repeatedly specify the relationship of that variable to these operators. The SETL algorithm for the static checking procedure described above is as follows:

```
definef staticval (node);

/* check whether a previously evaluated sub-tree was found
                                                    invalid */

if valid eq false then return om ;;
if atom(node) then        /* it is a lexical type */
return proptab(node);;
if prefixtab(node(1) is opname) ne om then
                                  /* it is a prefix operator */
return validpref (opname, staticval(node(2)));
else if infixtab(opname) ne om then /* it is an infix operator */
return validinf(opname, staticval(node(2)), staticval(node(3)));
else if opname eq index then
return validindex  (proptab(node(2)), staticval(node(3)));
else if opname eq ifstat then
```

```
        return staticval(node(2));
    else if opname eq call then          /* procedure call */
        arglist = nult;
        ( 1 ≤ k ≤ # (node(3) is listargs)) arglist = arglist +
                                    <staticval(listargs(k))>;
        return validproc (node(2), arglist);
    else if opname eq  ≠ = ≠ then      /*assignment statement */
    /* if the left-hand side is the name of a global variable, for
    which there may be no entry in proptab, then there is no
    consistency check to be made.  Rather, an entry into proptab has
    to be made for that variable */
        if (atom(node(2) is leftvar)) and (proptab(leftvar) eq om)then
            proptab(leftvar) = staticval(node(3));
        else if n ( ∃ valleft  ε staticval(node(2)),valright ε
                                                staticval (node(3))
                    |valleft eq valright) then
        print ( ≠ invalid assignment statement. types of I.h.s. and r.h.s.
                  are incompatible ≠);
        return om;
    else   return  t ;;
else                    /* it is not an executable statement */
return t;;
end staticval;
```

The validating procedures *validinf*, *validpref* and *validindex* have very similar forms. Each uses its own table; if an invalid operation is detected, each outputs a diagnostic message and sets the global flag *valid* to *false*.

```
definef validpref (prefixop, arg);
if n ( ∃ val  ε arg |(prefixtab(prefixop,val) is result) ne om)
     then valid = f;
     print (≠ invalid use of a prefix operator or field extractor ≠,
                    prefixop, arg);
     else return { result };;
end validpref;
```

similar code will appear, with obvious modifications, in *validinf* and *validindex* *Validproc* is somewhat more complicated. For each variable in the list of calling parameters, and for each of the global variables that are input to the subroutine, we try to find a match between the declared variable-types and the calculated *staticval'* s. If matching is sucessful, entries are made into *proptab* for all output parameters and global variables.

```
definef valid proc(procname, arglist);
/* arglist contains the calling parameters of the procedure.  In
the case of a subroutine this will include the output parameters.
We assume that arglist is ordered so that all input parameters
appear first. */
     inlist = proctab(procname)(1);
     invars = # inlist;
     ( 1 ≤∀k ≤ invars)
          if n (∃ type ε arglist(k)|type eq inlist(k)) then go to error;;;
     /* check global variables used for input. */
     globsin = proctab(procname)(3);
     ( ∀ globin ε globsin)
```

```
if n (∃ type ε proptab(globin(1))|type eq globin(2)) then go to error;;;
/* assign variable-types to output parameters. */
   outlist = proctab(procname)(2);
   (1 ≤∀k ≤ # outlist)
      proptab (arglist(k + invars)) = {outlist(k)};;
   /*  assign variable-types to global variables affected */
         globsout = proctab(procname)(4);
         ( ∀  globout ε globsout)
              proptab (globout(1)) = {globout(2)};;
    /* return the value of the function. If the procedure is a
subroutine, the returned value is superfluous,  as no further
                                          checks will be made on it. */
     return {outlist(1)};
     error:  print ( ≠ invalid parameters in procedure call ≠,
                              procname, arglist);

            ∀alid = f;
            return om;
end validproc;
```