SETL Newsletter # 71 April 17, 1972

Deducing the logical structure of objects occurring in SETL programs.

J. Schwartz

One of the next main problems to be faced in our work on SETL is that of optimization. This Newsletter, which continues Dave Shield's Newsletter 45, is intended to revive and carry forward discussion of this problem.

I shall concentrate on a particular aspect of the overall optimization problem,that of discovering, at compile time, the logical 'type' or 'structure' of the objects occurring in a SETL program. More specifically, we would like to be able to predict the type (e.g. integer; string, etc.) of atoms and the structure (e.g. set of strings; sequence of integers; triple of string, integer, and set of strings; etc.) of compound objects This information, if available, would permit many time-consuming type-checks to be bypassed. If combined with information concerning the operations applied to particular objects within a program, it might in some cases make it possible to choose special data layouts automatically. The overall lines of an attack on the type-analysis problem will be sketched below, under the simplifying assumption that sub-routines are not transmitted as arguments in the code being analyzed; this assumption enables us to view the flow structure of a program in an entirely static, rather than in a partly dynamic manner.

Main subheadings:

1. A representation of the flow and operation structure of a SETL program.
2. Detailed resolution of variable names.
3. A lattice of variable structures, and the effect of SETL operations on the elements of this lattice.
4. A global structure-predicting algorithm.

1. A representation of the flow and operation structure of a SETL program.

Our aim is to relate each SETL program P to an interpretable graph involving operations on data structures, and to analyze this graph, deducing the type/structure of each data item appearing in P. For this purpose, we use a somewhat modified program flow graph made up of basic blocks connected by a successor relation. To each block there will belong a sequence of operation items derived from the operations present in the section of code which the block represents. A block B will be a successor of a block $\overline{B}$ if $\overline{B}$ terminates in a conditional (or calculated) transfer which might have B as target. Read statements, iterations over sets, and subroutine calls require special consideration. First as to subroutine calls. We consider each call

```
sub (a, b+c, d, ...);
```

as an unconditional transfer to the subroutine, preceded by a set of assignments

```
arg1 = a; arg2 = b+c; arg3 = d; etc.
```

and followed by a labeled second set of assignments

```
label: a=arg1;  d=arg3;  etc.
```

A return statement within the subroutine sub is then treated as conditional or calculated transfer having as target each of the labels affixed in this way to a 'point of return' from sub. Functions are treated similarly, a function invocation f(a1,...,an) being regarded as a subroutine call subf(result,a1,...,an) preceded by the initialization

```
result = Ω;  .
```

A function return

```
return a;
```

is then regarded as being equivalent to

```
result = a;  return;
```

Next consider iterations over sets. We assimilate each iteration

(∀x ∈ a) block;

to the following less specific code, which has an equivalent effect on all variable types:

```
        x = ∋a;
cont:   if <indefinite> go to quit;
        block;
        go to cont;
quit:
```

where the conditional transfer shown has both the first statement of block and the label quit as possible targets. A set-former {e(x), x ∈ a | C(x)} is reduced to the explicit iteration

temp=nl; (∀x ∈ a | C(x)) temp = temp with e(x);;

which defines it; and similar reductions are made for existential and universal quantitifiers. An iteration of the form (k ≤ ∀n ≤ m) block; is reduced to

```
        n = some integer;
cont:   if<indefinite> go to quit;
        block;
        go to cont;
quit:
```

Read statements, which can hardly be treated in any other way, I propose to treat declaratively, i.e., by attaching to each read statement a declaration giving, in the form to be explained below, the structure of the item which the statement will read.

These conventions allow us to represent any SETL program by a graph consisting of blocks containing SETL primitive operations, and terminated by a transfer to some indefinite one of a group of possible successors. No subroutine calls, read statements, or explicit iterations will remain in this abstracted

representation of a program. The primitive operations which can occur are:

+, -, *, /, //

ε, eq, gt, representing all operations producing boolean values

max, min, abs

n, and, representing boolean operations

∋, with, less (which stands also for lesf)

pow(a), npow(k,a), hd, tl .

explicit tuple- by enumeration and set-by-enumeration operations

indexing: f(a), f(a,b) etc. As will be seen below, the case of a constant index is treated in a somewhat different way from the case of a variable index.

indexing in its second form: f{a}, f{a,b}, etc. The operations f[a], f[a,b] are treated as if they were f{∋a}, f(∋a,∋b), etc.

The indexed assignment f(x) = a is treated as if it were f = f less ∋f with <x,a>;

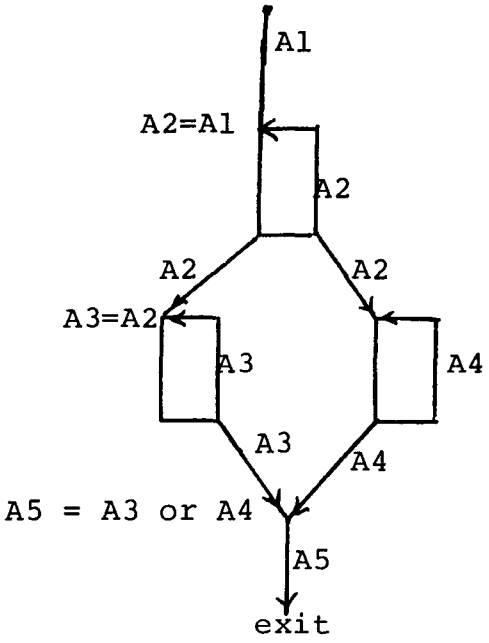
The indexed assignment f{x} = a is treated as if it were f = f less ∋f with <x,∋a>;

and the other forms of indexed assignment are treated in corresponding fashion.

Multiple assignments are broken down to several simple assignments.

2. Detailed resolution of variable names.

Our overall problem is to track the effect on data-object-type of each of these primitive operations. We will do this by associating a calculated object-type with each variable name occurring in a program. Of course, names of similar form occurring in disjoint name-scopes are to be treated as distinct. We will in fact wish to make an even finer resolution of names than this, treating identical names as distinct even if they occur in the same name scope, provided however that the variables they designate are never 'live' at the same time. The following specific rule is proposed: Let A be a variable name. A flow graph-edge which is part of a path leading to a use of A which is not preceded by an assignment to A is an edge along which A is live. Consider the subgraph G_A of the program graph G which includes only those edges along which A is live. The strong-components (maximal strongly connected subregions) of G_A are called the A-components of G. We introduce an individual resolved name in place of A for each such A-component and logically identify each strong component G_A to a point. This reduces G_A to a loop-free graph. Each occurrence α of a name A along an arc of G_A not belonging to a strong-component of G_A is translated into an occurrence of a corresponding resolved name A_j according to the following rule: if the edge containing α is a successor of exactly one strong-component C of G_A, then A_j is the resolved name associated with C. If the edge containing α is a successor of several strong components, within which A has been assigned resolved names A_j, A_k, A_ℓ, ..., then on the edge A is assigned the name A_j or A_k or A_ℓ. More precisely, we introduce yet another name A', and proceed as if assignments A' = A_j; A' = A_k; etc. had been encountered. The naming scheme that results is illustrated in the following figure.



3. A lattice of variable structures, and the effect of SETL operations on the elements of this lattice.

Next we outline the lattice Λ of structure-representing elements which we shall use to describe the nature of variables occurring in SETL programs. This lattice contains 'maximal' elements representing particular types of atoms, and a unique 'minimal' element ω representing the lack of all compile-time knowledge of a variable's structure. The 'type' ω may also be thought of as corresponding to the 'general' SETL object, which may be an arbitrary atom, set, or tuple. Elements of Λ which are sufficiently complex will be identified with ω; that is, it is only when the type of SETL variables remain simple that we will attempt to follow these types in detail.

Elementary types are:

Ω (type of undefined atom); I (integer); C (character string);
B (boolean, i.e., true or false); BB (boolean string);
N (nullset); T (null-tuple); ω (undefined type)

Compound types are formed from simpler types using the following operations.

$t_1 \mid t_2$	-	type alternation, t_1 'or' t_2
$\{t_1\}$	-	set whose elements are of type t_1
$<t_1,t_2,\ldots,t_k>$	-	type of known length whose components are of known types $t_1,t_2,\ldots,t_k$
$[t]$	-	tuple of indefinite length whose components are known to be of type t.

Only the alternation of elementary types will be carried explicitly. Other alternations will be reduced to simpler, and generally less explicit, forms using the following rules:

$\{t_1\} \mid$ elementary type $\rightarrow \omega$, except that $\{t_1\} \mid N \rightarrow \{t_1\}$

$\{t_1\} \mid <t_2,\ldots> \rightarrow \omega$

$\{t_1\} \mid [t_2] \rightarrow \omega$

$\{t_1\} \mid \{t_2\} \rightarrow \{t_1 \mid t_2\}$

$<t_1,t_2,\ldots,t_k> \mid <t_1',t_2',\ldots,t_k'> \rightarrow <t_1 \mid t_1', t_2 \mid t_2', \ldots>$
for type-tuples of same length

$<t_1,t_2,\ldots,t_k> \mid <t_1',t_2',\ldots,t_j'> \rightarrow [t_1 \mid t_2 \mid \ldots \mid t_k \mid t_1' \mid \ldots \mid t_j']$
for type-tuples of different length

$<t_1,\ldots,t_k> \mid$ elementary type $\rightarrow \omega$

$<t_1,\ldots,t_k> \mid [t] \rightarrow [t_1 \mid t_2 \mid \ldots \mid t_k \mid t]$

$[t] \mid$ elementary type $\rightarrow \omega$, except that $[t] \mid T \rightarrow [t]$

$[t_1] \mid [t_2] = [t_1 \mid t_2]$.

Using these rules, alternation signs can always be moved to 'innermost' position. Note also that $t \mid t = t$.

Next, we introduce the rule that 'bracketing' in a type structure may not be carried more than three levels deep. Higher degrees of nesting are removed by replacing the most deeply nested substructures by ω. Thus, for example,

$$< < < <t1,t2>, t3>, t4>, t5>$$

becomes

$$< < <\omega,t3>, t4>, t5> .$$

Each elementary SETL operation acts in a certain way on our algebra of types. Salient details are as follows for the case of the operator '+' (which typifies one class of SETL primitive)

$\Omega + \Omega$ = diagnostic; I + I = I; C + C = C; B + B = BB ;

BB + BB = BB; N + N = N; T + T = T; $\omega + \omega = \omega$;

all other sums of elementary types give diagnostics, except

B + BB = BB; I + ω = I; C + ω = C; B + ω = BB;

BB + ω = BB; {t} + ω = {ω}; [t] + ω = [ω]; etc.

$<t_1,t_2,\ldots,t_k> + \omega = [\omega]$.

The rule for alternating types is as follows:

$(t_1|t_2) + (t_3|t_4)$ = alternation of all those combinations

t_1+t_3, t_1+t_4, t_2+t_3, t_2+t_4 which do not give diagnostics.

For sets, tuples, and sequences we have:

$<t_1,t_2,\ldots,t_k> + <t_1',t_2',\ldots,t_j'> = <t_1,t_2,\ldots,t_1',\ldots,t_j'>$;

$\{t_1\} + \{t_2\} = \{t_1|t_2\}$;

$[t_1] + [t_2] = [t_1|t_2]$;

$<t_1,t_2,\ldots,t_k> + \{t\}$ = diagnostic;

$<t_1,t_2,\ldots,t_k> + [t] = [t_1|t_2 \ \ldots \ t_k|t]$;

$[t_1] + \{t_2\}$ = diagnostic;

{t} + elementary = [t] + elementary = $<t_1,\ldots>$ + elementary

= diagnostic,

unless the elementary is N or T

Comparison operators like eq, gt, etc. always return the value B ; the membership test returns B if its second argument is {t}, and a diagnostic otherwise. Operations in general are distributive over the alternation sign |, with elimination of 'impossible' cases, i.e., cases which would lead to the issuance of diagnostics.

I continue to list the effect of the primitive SETL operations on the family of data types which have been introduced:

$\exists[t]$ = diagnostic; $\exists\langle t_1,\ldots\rangle$ = diagnostic; $\exists\{t\} = t$;

$[t_1]$ with t_2 = diagnostic; $\langle t_1,\ldots\rangle$ with t_2 = diagnostic;

$\{t_1\}$ with $t_2 = \{t_1|t_2\}$;

less behaves similarly, except that $\{t_1\}$ less $t_2 = \{t_1\}$;

pow([t]) = diagnostic; pow($\langle t_1,\ldots\rangle$) = diagnostic;

pow({t}) = {{t}}; and similarly for the npow function.

Indexing:

$[t_1](I) = t_1|\Omega$; $[t_1](t_2)$ = diagnostic if t_2 is not I.

$\langle t_1,\ldots,t_k\rangle(I) = t_1|t_2|\ldots|t_k|\Omega$, unless I = j is a constant,

in which case $\langle t_1,\ldots,t_k\rangle(N) = t_j$.

$\{\langle t_1,t_2,t_3,\ldots\rangle\}(t_1) = \langle t_2,t_3,\ldots\rangle|\Omega$;

$\{[t_1]\}(t_1) = [t_1]|\Omega$;

$\{\langle t_1,\ldots\rangle\}(t_2) = \Omega$ if $t_1 \neq t_2$.

$\{[t_1]\}(t_2) = \Omega$ if $t_1 \neq t_2$.

In most other cases these operations give a diagnostic. Similar rules apply for functional application to several arguments and to functional application in its second form. For example,

$\{\langle t_1,t_2,t_3,\ldots\rangle\}\{t_1\} = \{\langle t_2,t_3,\ldots\rangle\}$;

$\{\langle t_1,t_2,t_3,\ldots\rangle\}\{t\} = N$ if $t \neq t_1$, etc.

Corresponding rules for $\{t_1\}[t_2]$, etc. are derivable from these. For example, $\{t_1\}[t_2]$ is $\{t_1\}\{\ni t_2\}$.

The rules which have been stated allow a sequence of type-calculations to be associated with each basic block of a SETL program. We now turn to discuss the manner in which these 'elementary' type-calculations are integrated to allow a type to be computed for each resolved variable name occurring in an entire program.

4. A global structure-predicting algorithm.

We intend, unless this proves to be hopelessly inefficient, to compute overall types using a rather naive process which goes through the abstract flow-graph version of a program iteratively and repeatedly, assigning less and less 'specific' types to each variable until this process is stabilized. More precisely, we start with the first block of a program, and taking the type of each constant (and of each data item read) to be known. The type U ('uninitialized') is at the start of our processing associated with each variable name. Then each calculation is taken up in sequence, and yields an object of calculable type (which may, of course, be the completely indefinite type ω). An assignment of a quantity of a type t_2 to a variable A with which the type t_1 is associated will give A the value $t_1|t_2$. We work our way through all the blocks of the program graph in turn, maintaining a workpile of blocks to be processed. As a block is processed, we note any change (always from more to less specific) in the types associated with the variables occurring in it. If the processing of a block works any such change, we must on completing the block add all its successors to our workpile, since the information associated with variables occurring in these blocks may require revision.

Sketch-algorithms, written in SETL, are as follows:

```
define typeprocess (pgraph);
/* the main 'driver' of the type-analysis process */
<cesor, ops, head> = pgraph;
/* 'cesor' is the successor map defining the program flow.
  we take each node to be a blank atom, and 'ops' to be a
  map sending each such node to a tuple representing the
  operations in the block represented by the node */
work = {head}; (while work ne nl) node from work;
modif = blockprocess(ops(node));
/* the routine blockprocess works its way through the successive
  operations of a block, updating the type of each variable
  occurring in the block. if any type is thereby modified,
  the value t is returned; otherwise the value f */
if modif then work = work + cesor(node);;
end while; return;
end typeprocess;
definef blockprocess(optuple);
/* the components of optuple are themselves tuples, representing
  the primitive operations of a SETL program in the following way.
  monadic ops: <result variable, opcode, input>
  binary ops: <result variable, opcode, input1, input2>
  ops with more inputs: <result variable,opcode,input1,...> */
modif = f;
(1 ≤ ∀n ≤ #optuple) <result,opn> = optuple(n);
if newtype(opn) is newt eq 'd' then continue; else
type(result) = newt alt (type(result) is oldtype);;
/* newtype(opn) calculates the type of an operation result
  from knowledge of the operation and the type of all its inputs*/
/* 'alt' reduces to canonical form the 'alternation' of a variable's
  former type and the result-type calculated by newtype */
if type(result) ne oldtype then modif=t;;
end ∀n; return modif;
end blockprocess;
definef t1 alt t2;
/* this routine embodies those rules for calculating type
  alternations which were stated in section 3 */
```

```
        /* the encoding used for types is as follows:
          elementary types are represented by the characters
          I, C, N, T, B, A (for BB), O (for Ω), U (for ω), and
          D (for impossible or 'diagnostic' type);
          an alternation of elementary types is represented by a
          string concatenation of these characters */
        /* the type '{t} is represented by the set whose only element
          is the object representing the type t;
          in the same sense, <t> represents the type [t],
          and <t1,...,tn,0> represents the type <t1,...,tk> */
        go to '{<str,elem>, <set,setc>, <tupl,tupc>}(type t1);
  setc: if type t2 eq set then return '{∋t1 alt ∋t2} else if t2 eq N then
                return t1;  else return 'u';
  tupc: if #t1 gt 1 then go to truetup;;
                /* otherwise a sequence is represented */
        if type t2 eq tupl then go to seqtup;
                else if t2 eq 't' then return t1;
                   else return 'u';;
 seqtup: if #t2 eq 1 then return < hd t1 alt hd t2>;
           else return  <hd t1 alt [alt: 1≤ n< #t2]t2(n)>;;
truetup: if type t2 eq tupl then go to tuptup;
           else if t2 eq 't' then
           return <[alt: 1≤ n< #t1] t1(n)>;
           else return 'u';;
 tuptup: if #t2 eq #t1 then
           return [+: 1≤ n≤ #t1] <t1(n) alt t2(n)>
           else return  <[alt: 1≤ n< #t1] t1(n) alt
           [alt: 1≤ n< #t2] t2(n)>;
   elem: if n type t2 nl str  then return t2 alt t1;
           else set = '{t1(n), 1≤ n≤ #t1} + '{t2(n), 1≤ n≤ t2};
           return [+: c ∈ set]c;;
        end alt;
```

```
        definef newtype(opntup)
        /* this function receives as input a tuple consisting of an
           operation sign and its arguments, and produces as output
           the type of the operation result. if the stated inputs
           are invalid for a particular operation, the special output
           'd' = diagnostic type is produced */
        /* note that this routine embodies a variety of special observa-
           tions  like those set forth in the second part of section 3.
           we shall not give the whole of the rather miscellaneous code
           that this routine requires, but only a few suggestive fragments*/
        <op,args> = optup;  /*sort out particular operations involved */
        go to {<pls,plscas>, <mins,minscas>, <tms, timscas>,
        <div,divcas>,< mod,modcas>,< max,maxcas >, ...}(op);
        /* here follows sample code for treatment of the 'plus' operation*/

plscas: <a1,a2,-> = args;
switch: go to {<set, plset>, <tupl,pltup>, <str,plelem>}(type a1)
 plset: if type a2 eq set then return {  a1 alt   a2};
           else if a2 eq 'n' then return a1;  else if a2 eq 'u'
           then return 'u' else return 'd';;
 pltup: if a2 eq 't' then return a1; else if a2 eq 'u' then return 'u';
           else if type a2 ne tupl then return 'd';
           else if (#a1) gt 0 a(#a2) gt 0 then return
            a1(1: #a1-1) + a2(1: #a2-1) + <0>;
           else return <[alt: 1≤ n≤ (#a1-1) max 1] a1(n) alt
                        [alt: 1≤ n≤ (#a2-1) max 1] a2(n)>;
           end if;
        /* enter here if first argument is elementary */
plelem: if (type a2) ne str then <a2,a1,-> = <a1,a2>;
           go to switch;;
        /* otherwise have two elementary arguments */
        return elbin(pls, a1, a2);;
        ... /* and so forth. the routine elbin computes the type
                 of a combination of elemtnary types */
        ...
        end newtype;
```

```
definef elbin(op,a1,a2);
/* computes the type of a binary combination of elementary types */
set = {if elbintab(op,a1(n),a2(m)) is ebt ne Ω then
        ebt else 'd' | 1 ≤ n ≤ #a1, 1 ≤ m ≤ #a2};
if set ne {'d'} then set = set less 'd';;
return [+: c ∈ set]c;
/* the table 'elbintab' gives the type of every possible valid
   combination of elementary operands. the entries pertaining
   to the 'pls' operation are as follows */
/* <pls,I,I,I>, <pls,C,C,C>, <pls,B,B,BB>,
   <pls,N,N,N>, <pls,T,T,T>, <pls,B,A,A>,
   <pls,A,B,A>, <pls,U,I,U>, and so forth */
/* in the actual implementation, a better (denser) multi-level
   encoding of this information should of course be devised */
end elbin;
```

Experimentation with the algorithm in the form sketched above should show whether it attains an acceptable efficiency. If not, a revised version making use of 'use-definition chaining' much in the manner that has been proposed for the constant propagation process might be more efficient.

Note also for future use that this algorithm begins to reveal something of the process that should be used in knitting together SETL programs in the presence of data structure elaborations. Such similarity between the problem discussed in this newsletter and the more difficult problems connected with data structure elaboration comes from the fact that the manner in which a variable's value is 'encoded' can be regarded as constituting a generalized 'type' attaching to the variable.