SETL Newsletter # 121 January 2, 1974

An Algorithm to Determine the Identity of SETL Run-Time Objects

Aaron Tenenbaum

One of the causes of SETL's current inefficiency is its method of storing all sets in a way which facilitates function application. If a set is not used as a function, a more suitable storage and retrieval scheme for it would cause an increase in efficiency. This consideration leads to the desirability of an algorithm to determine at compile time how a particular set is being used, which would in turn determine the most efficient storage method for that set. Ordinarily, this would be a simple matter of use-definition chaining except for the following complication. Assume we have code of the form:

```
f = f with s;
t = arb f;
```

Since the with operation involves only inserting a pointer to the element into the hashtable of the set and the arb operation simply involves retrieval of that pointer, the variables s and t might refer to the same run time object. Thus any operation applied to t would be relevant in determining a storage and retrieval method for s. The same problem occurs in the sequence:

```
f(n) = s;
t = f(n);
```

The identification of s and t in the above examples will be called a *first-level identifcation.*

The situation may be even more complex such as in determining the identify of x and y in:

```
s = s with x;
f = f with s;
t = arb f;
y = arb t;
```

Such an identification, depending as it does on a previous first-level identification (in this case, t and s) will be called a *multi-level identification.*

Our first focus of attention is to compute the function *contents(b,x)* which for each basic block b and variable x would yield the set of all definitions in the program whose

defined variable may be an element of x upon entry to b. The method of computing this function follows closely the method of computing available computations upon entry to a block in the redundant computation detection algorithm.

We assume given the following functions which can easily be computed by analyzing each basic block:

inserted(b,sb,x) — for all basic blocks b, sb a successor of b, and variable x, the set of all definitions whose defined variable has been inserted into the item represented by x along the path through b to sb such that x has not been redefined along that path after the point of insertion (except possibly by a with, less, lesf operation on itself, or an indexed assignment to itself).

clear(b,sb) — for all basic blocks b, sb a successor of b, the set of all variables which have not been redefined along the path through b to sb (again, with the above exceptions).

If we already knew *contents(pb,x)* for every predecessor block pb of b then we could compute *contents(b,x)* using the equation:

$$\text{(1)}\quad \text{contents(b,x)} = \bigsqcup_{pb \in \text{pcesor(b)}} (\text{inserted(pb,b,x)} \cup \text{if } x \in \text{clear(pb,b)} \text{ then contents(pb,x) else } \emptyset)$$

where *pcesor* is a map from each node to the set of its predecessors. Therefore, if we knew *contents(int,x)* for each interval *int*, we could then process the basic blocks which make up the interval in interval order to compute their *contents* sets.

Two extreme assumptions can be made about *contents(int,x)*: one is that all defined variables are elements of x, and the other that none are. We can define two functions on the basic blocks which make up the interval under these two assumptions:

posc(b,x): the contents of x at entry to b under the assumption that all defined variables are elements of x at entry to the containing interval.

defc(b,x): the contents of x at entry to b under the assumption that no defined variable is an element of x at entry to the containing interval.

Once these two functions have been defined, and given *contents(int,x)*, the calculation of *contents(b,x)* is a simple one and is given by:

(2) contents(b,x) = (contents(int,x) ∩ posc(b,x)) ∪ defc(b,x)

When the graph is reduced to a single interval, *int*, the quantity *contents(int,x)* is known to be null, so that we can proceed backwards in the derivation sequence to calculate *contents(b,x)* until we come down to the level of basic blocks.

Our only remaining task, therefore, is to compute *posc* and *defc* for intervals. To do this, we use the analogs of equation (1), substituting *posc* and *defc* for *contents*. However, equation (1) necessitates knowing *clear* and *inserted* for intervals. Letting preds = pcesor(sint(1)) ∩ {int(i), 1 <= i <= #int} we have that:

(3) inserted(int,sint,x) = $\underset{pb\in preds}{\lfloor\_\rfloor}$ (inserted(pb,sint(1),x)
if x∈clear(pb,sint(1)) then defc(pb,x) else ∅)

To compute *clear(int,sint)*, we have to face the possibility that there may be more than one exit from *int* to *sint*. Since we want to compute all possible contents of each variable, we must assume that *clear* is as large as possible. This leads to the calculation of *clear* along the path terminating at the earliest possible exit from *int* to *sint*. Letting *ind(b,int)* be the index of b in *int* we have:

(4) blnum = $\underset{b\in preds}{\min}$ ind(b,int)

clear(int,sint) = $\left(\underset{1<=i<\ blnum}{\lceil\_\rceil}\ clear(int(i),int(i+1))\right)$ ∩
clear(int(blnum),sint(1))

Thus the entire process involves two passes: the first going in the order of the derivation sequence, computing *clear* and *inserted* for an interval using equations (3) and (4) and their previously computed values for the constituent nodes of the interval while simultaneously using the analogs of equation (1) to compute *defc* and *posc* for those constituent nodes; and the second pass proceeding in reverse order of the derivation sequence, using equation (2) to compute *contents*.

The driving routine follows.

```
define contdrv(seqd);
   /* contents and vars are global.  vars is the set of all
      variables in the program, seqd is the derivation sequence*/
   /* proceed along the derivation sequence in the first pass*/
   (1 < ∀k <= #seqd), intv ∈ hd(seqd(k))) contprl(intv);;
   contents = nl;
   /* initialize contents for every variable to nl at the entry
      to the single interval to which the graph eventually reduces*/
   (∀x∈vars) contents(arb hd (seqd(#seqd)),x) = nl;;
   /* proceed in reverse derivation sequence for the second pass*/
   (#seqd >= ∀k > 1, intv ∈ hd(seqd(k))) contdef(intv);;
   return;
end contdrv;
```

The following routine is called for each interval in the first pass to compute *posc* and *defc* for the constituent blocks of the interval, as well as *inserted* and *clear* for the interval itself.

```
define contprl(intv);
   /* vars, defs, posc, defc, clear, pcesor, cesor and inserted
      are global. defs is the set of all definitions in the
      program, cesor is the map from each block to the set of
      all its successors */
```

```
   hed = intv(1);      defc = nl;      posc = nl;
   nodesof = {intv(i), 1 <= i <= #intv};
   (∀x∈vars) defc(hed,x) = nl;          posc(hed,x) = defs;
      (1 < ∀i <= #intv) preds = pcesor(intv(i) is b);
         /* use analogs of equation (1) */
         defc(b,x) = [+: pb∈preds] transf(pb,b,defc,x) orm nl;
         posc(b,x) = [+: pb∈preds] transf(pb,b,posc,x) orm nl;
      end ∀;
   end ∀;
   (∀sint ∈ cesor(intv)) preds = pcesor(sint(1) is sl) * nodesof;
      (∀x ∈ vars) /* use equation (3) */
         inserted(intv,sint,x) = [+: pb∈preds]transf(pb,sl,defc,x)
                                                      orm nl;
      end ∀;
      /* use equation (4) */
      blnum = [min: b ∈ preds] ind(b,intv);
      clear(intv,sint) = ([*: 1 <= i < blnum] clear(intv(i),
            intv(i+1)) orm nl) *clear(intv(blnum),sl);
   end ∀;
   return;
end contprl;
```

This routine uses the following auxiliary routines:

```
definef  a orm b;  return if a ne Ω then a else b;  end;

definef ind(elt, tuple);
   /* finds the location of an element in  a tuple */
   (1 <= ∀i <= #tuple) if elt eq tuple(i) then return i;;;
   return 0;
end ind;

definef transf(pb,b,tset,x);
   /* inserted and clear are global */
   return(inserted(pb,b,x) + if x∈clear(pb,b) then tset(pb,x)
                                          else nl);
end transf;
```

The next routine is used on the second pass to compute the final value of contents:

```
define  contdef(intv);
   /* vars, contents, posc and defc are global */
   (∀x ∈ vars, b(i) ∈ intv) /* use equation (2 ) */
      contents(b,x) = (contents(intv,x) * posc(b,x)) + defc(b,x);
   end ∀;
   return;
end contdef;
```

This concludes our discussion of how to derive *contents*, and we now turn to an algorithm for determining the identification of distinct variables. First, however, we must discuss how the program is represented in these algorithms. The mapping *progrph* assigns to each block in the program, a tuple of instructions occurring in the block in order of occurrence. Each instruction is represented by a tuple, the first element of which is the output variable, the second the opcode and the remaining elements are the input variables. Thus the instruction x = y + z may be represented by <*'x'*, *oad*, *'y'*, *'z'*>, where *oad* is the macro whose value is the opcode for the plus operation. Some opcode macros which will appear in the algorithms below are:

owth	the SETL with operation
olss	the SETL less operation
olsf	the SETL lesf operation
oset, *otpl*	the set and tuple formers
oof	functional application or indexing a tuple
ondxass	indexed assignment.

This last operation of indexed assignment, takes n+2 input variables if n indices are present. The first n are the indices, then comes the quantity to be assigned and last the set or tuple into which one is indexing. This set or tuple

also appears as the output variable. Note that if we perform an indexed assignment to a tuple the instruction tuple will be of length 5 (tuple name, opcode, index, value, tuple name), and if the length is greater we know that we are indexing a function of more than one variable. Similarly an indexed load from a tuple has length 4 (result, opcode, tuple name, index).

A definition is represented as a triple consisting of the defined variable, the block in which the definition occurs and the location within the block of the definition. Similarly, a use is represented by a quadruple consisting of the variable used, the block in which the use occurs, the location of the instruction within the block and the location of the use among the input variables of that instruction. For example the instruction tuple *<'x', oad, 'y', 'z'>* occurring as the fourth instruction in block 7 would give rise to the definition *<'x',7,4>* and the uses *<'y',7,4,1>* and *<'z',7,4,2>*. We also assume the existence of two functions computed by a use-definition chaining algorithm: *ud*, which given a use returns the set of all definitions to which that use may refer, and *du*, which given a definition returns the set of all uses which may utilize the quantity being defined.

The auxiliary routine *oper*, given a use or definition returns the opcode of the instruction in which it appears:

```
definef oper(a);
   /* progrph is global */
   return ((progrph(u(2)))(u(3)))(2);
end oper;
```

The routine *defn*, given a use returns the output variable of the instruction in which the use exists:

```
definef defn(u);  /* progrph is global */
   return ((progrph(u(z)))(u(3)))(1);
end defn;
```

With these preliminaries taken care of, we can turn to the algorithm itself. We construct a mapping *equiv* which assigns to each definition in the program the set of all definitions occurring prior to it on some execution path which may define the same run time object.

The routine *seteq* initializes *equiv* by searching for uses of the definition which result in a modification of a run time object by insertions or deletions. Such uses are those occurring in with, less, lesf operations in which the defined variable is the same as the first input variable or in indexed assignment operations. For example, given the sequence

```
x = . . .
x = x with y ,
```

seteq would add the first definition of *x* to *equiv* of the second. The code for seteq is:

```
define  seteq;
   /* progrph, du, equiv, and defs are global */
   equiv = nl;  st = {owth, olss, olsf, ondxass};
   (∀d ∈ defs) equiv(d) = nl;;
   (∀d ∈ defs) total={u(1:3), u∈du(d)|oper(u)∈st and
                        defn(u) eq u(1)};
      (∀df ∈ total) doequiv(df,{d});;
   end ∀;  return;
end seteq;
```

The above uses a routine *doequiv* which accepts a definition *d* and a set of definitions, *total*. It adds *total*, *equiv*[*total*], *equiv*[*equiv*[*total*]], etc. to *equiv(d)*. Further, if *d* ∈ *equiv(df)*, it also adds all of these sets to *equiv(df)* and similarly to *equiv(def)* if *def* ∈ *equiv(df)*, etc. The code for *doequiv* is:

```
define  doequiv(d,total);
  /* defs and equiv are global */
  equiv(d) = equiv(d) + total;
  work = total;   sum = total;
  (while work ne nl) df from work;
     equiv(d) = equiv(d) +(equiv(df) is edf);
     work = work + edf;  sum = sum + edf;
  end while;
  work = {d};
  (while work ne nl) df from work;
     (∀def ∈ defs | df ∈ equiv(def))
        equiv(def) = equiv(def) + sum;
        def in work;
     end ∀;
  end while;
  return;
end doequiv;
```

After *equiv* has been initialized, we process the program to discover variables which may refer to the same run time object by first-level identifcation. The method used is as follows: for every definition which involves an arb operation on a set or a possible indexed load from a tuple, search through all operations which precede it in its basic block for indexed assignments to the tuple or variables placed in the set by a with operation, or variables placed in the set or tuple at creation with a set-former or tuple-former operation. Such variables may be identified with the currently defined variable. This backwards search continues through the basic block until a definition is found in which the set or tuple is redefined using an operation other than less, lesf, with or indexed assignment. Such a redefinition acts as a block to identification of the current variable with any prior definitions. If no such redefinition is found, *contents* of the variable representing the set or tuple at entry to the block is added to the set of definitions which may be identified with the one in question. The code to do this follows:

```
define eqproc;
   /* defs, progrph, contents, and ud are global */
   (∀d∈defs)optupl = progrph(d(2)); /* the tuple representing
                                       the basic block */
      op = optupl(d(3)); /* the tuple representing the instruction*/
      if op(2) eq oarb or(op(2) eq oof and #op eq 4) then acc=nl;
      /* acc is used as an accumulator set for all definitions
         which may be identified with the current one */
         fl = t;  /* fl = t if no redefinition blocks our progress */
         (d(3) > ∀i > 0) /* proceed backwards from the current
                                       definition in the basic block */
            opp = optupl(i);
            if opp(1) eq opp(3) then
               if (opp(2) is opr) ∈ {oset,otupl} then
               /* add all things placed in the set or tuple to the
                  set of identifiable definitions; also, since
                  this represents a redefinition, set  fl to false
                  and leave the iteration */
               acc = acc +
                  [+: 2<j<#opp] ud(<opp(j),d(2),i,j-2>) orm nl;
               fl = f; quit ∀i;;
               if (opr eq owth and opp(1) eq opp(3)) or
                  (opr eq ondxass and #opp eq 5) then
               /* add the variable placed in the set or tuple to
                  the set of identifiable definitions */
                  acc = acc + ud(<opp(4), d(2), i, 2>);
               else if opr ne olss and opr ne olsf then
               /* we have reached a redefinition; set fl to false and
               quit the iteration */
               fl = f;  quit ∀i;;
            end if;
         end ∀;
         /* if the set has not been redefined, add the contents
            at block entry */
         if fl then acc = acc + contents(d(2), op(3));;
         doequiv(d,acc);
      end if;
   end ∀;
end eqproc;
```

To take care of multi-level identifications, we accumulate all definitions involving an arb or an indexed load from a tuple in a set *arbdefs* as we execute the above algorithm. After the first-level pass described above is complete, we check whether any of the uses of a variable defined by one of the definitions in *arbdefs* appears in an arb or indexed load operation; if so, the variable defined by that operation is a candidate for multilevel identification with the currently defined variable. For example, in the sequence

```
(0)  x = ...
(1)  s = s with x;
(2)  f(n) = s;
(3)  w = w with s;
(4)  g = arb w;
(5)  t = g(n);
(6)  y = arb t;
```

arbdefs = {(4),(5),(6)}. Focusing on definition (4) which defines variable g, we note that there is indeed an indexed load operation from that variable in definition (5), so that the variable *t* appearing there is a candidate for multi-level identification. Similarly, definition (5) defines variable *t* which is used in definition (6) as the set upon which an arb operation is performed, so that the variable *y* defined by (6) is also a candidate for multilevel identification.

Such new identifications are made in the following manner. Let *x* be a variable "arb-ed" from *y*. We construct *total*, the set of all definitions which are to be identified with any variables "arbed" from *x*. For all definitions identified with the definition of *x*, check to see whether that definition involves inserting some variable *w* into the defined quantity. If it does, then add the definition of *w* to *total*.

For example, in the sequence (0)-(6) given above, the first-level pass will set *equiv*(4) = {(2)}. Choose a definition from *arbdefs*, say (4). We check on (2) which is in *equiv*(4) and note that (2) inserts the object defined by (1) into its

defined variable. Therefore the object defined by (1) and any objects "arbed" from the object defined by (4) are equivalent. Thus *equiv*(5) = {(1)}. A similar process on (5) leads to *equiv*(6) = {(0)}.

Note, however, that if we would have processed *arbdefs* in the above example in a different order, e.g. (5), (4), we would have only recognized that *equiv*(5) = {(1)} but not that *equiv*(6) = {(0)}. This is because we cannot make the identification of x and y until after s and t have been identified. For this reason, we must keep on processing all of *arbdefs* until none of the *equiv* sets have been changed in an entire pass.

The code for the entire algorithm is therefore as follows:

```
define eqproc:
   /* defs,contents, du, ud, equiv, and vars are global */
   arbdefs = nl;
   (∀d ∈ defs) optupl = progrph(d(2));   op = optupl(d(3));
      if op(2) eq oarb or (op(2) eq oof and #op eq 4) then
         acc = nl;  d in arbdefs;  fl = t;
         ( d(3) > ∀i > 0) opp = optupl(i);
            if opp(1) eq op(3) then
               if(opp(2) is opr) ∈ {oset,otpl} then acc = acc +
                        [+:2<j<=#opp]ud(<opp(j),d(2),i,j-2>)orm nl;
                  fl = f;  quit ∀i;;
               if (opr eq owth and opp(1) eq opp(3)) or
                  (opr eq ondxass and #opp eq 5) then
                  acc = acc + ud(<opp(4),d(2),i,2>);
               else if opr ne olss and opr ne olsf then
                  fl = f;  quit ∀i;;
            end if;
         end ∀;
         if fl then acc = acc + contents(d(2),op(3));;
         doequiv(d,acc);
      end if;
   end ∀;
   /* this is the end of the first-level processing; now comes
      the multi-level case */
```

```
SETL 121-13
    fℓ1 = t; /* fl1 is a flag indicating whether or not to continue*/
    (while fℓ1) fℓ1 = f;
       (∀d ∈ arbdefs|oper[du(d)] * {oarb,oof} ne nl)
          total = nl;
          (∀df∈equiv(d)) optupl=progrph(df(2)); op=optupl(df(3));
             if (op(2) is opr) ∈ {oset,otpl} then
                (2 < ∀j <= #op) if op(j) ∈ vars then
                   total=total+ud(<op(j),df(2),df(3),j-2>);;;;
             if opr eq owth or (opr eq ondxass and #op eq 5)
                then total=total+ud(<op(4),df(2),df(3),2>);;
          end ∀;
          (∀u ∈ du(d))
          /* we're looking for objects "arbed"  from the
             variable defined by d */
          optupl = progrph(u(2));  op = progrph(u(3));
          if (op(2) eq oarb or (op(2) eq oof and #op eq 4))
             and not (equiv(<op(1),u(2),u(3)> is df) incs total)
             then fl = t;  doequiv(df,total);;
          end ∀;
       end ∀;
    end while;
end eqproc;
```