SETL Newsletter # 118

A. Tenenbaum
October 20, 1973

Revised and extended algorithms for deducing the Types of Objects Occuring in SETL Programs.

In a language without declarations such as SETL, any variable may at any point in a program represent a value having one of several different data types. During execution, the type of the variables must be checked to determine the meaning of an operation. Of course, this is time consuming and accounts in some measure for SETL's inefficiency. When the type of a variable can be determined at compile time, a compiler can in principle produce code to perform the desired operation more efficiently.

Even if we do not insist on programmer declaration of all variables, the type of a variable can be determined at compile time in one of two possible ways:

i) If a variable $x$ is the result of operator *op* applied to quantities $y$ and $z$ of known type, the type of $x$ can be deduced by knowing what type of results *op* produces from objects of the types of $y$ and $z$. For example, if $x=y+z$ appears in a program and $y$ and $z$ are tuples, then $x$ must also be a tuple.

ii) The type of a variable can often be determined merely from the fact that a given operation is applied to it. For example, if $\underline{t\ell}\ x$ appears in a program, then $x$ is known to be a tuple.

There are two chief differences between these two methods of type determination. The first difference is that the first method propagates knowledge of types in the direction of execution flow while the second method propagates that knowledge in the reverse direction. The second difference is that when dealing with compound types such as sets and tuples, the first method will give much more detailed type information about the constituent elements within the compound type.

To illustrate these differences, consider the following two examples:

a) x = 2; <u>read</u> y;

z = y + x;

b) <u>read</u> x, y;

z = x + y; w = <u>tℓ</u> z;

In example a) $x$ is of known type integer since it results from an assignment operation on an integer constant. Although the type of $y$ cannot yet be determined, the type of $z$ is known to be integer since it results from adding an integer ($x$) to some quantity. Note that these deductions are an example of method (i) and that type information has propagated in the direction of program flow. However, once we know that $z$ is an integer, we see that the use of $y$ is in an addition which results in an integer, so that $y$ itself must be an integer. Therefore in the *read* statement where $y$ is defined, an integer must have been read in. This in an example of method (ii). Similarly in example (b) above, since $z$ is involved in a <u>tℓ</u> operation, it must be a tuple, once this has been determined, $x$ and $y$ can also be classified tuples by their use in a plus operation which produces a tuple.

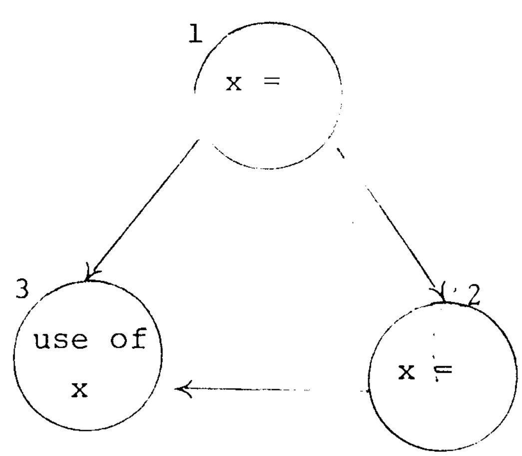
The deduction of types by method (i) is a relatively straightforward matter. If the types of all uses making up an operation are known, the deduction of the resulting type amounts to merely looking up in a precalculated table the type that will result when the given operation is applied to the given types.

However, type deduction by method (ii) is more complex. In this method, the type of a defined variable is deduced from the way in which the variable is used in subsequent operations. Thus we must look into the question of how the type of a variable use can determine the type of that variable at definition.

Specifically, if two uses of a variable exist along two disjoint paths of control flow from the definition, we cannot say that the definition must have the types of both uses since the branch may have been dependent on the variable's type and may have been specifically designed to bypass one of the uses when the variable's type is incompatible with that of the use. However, if two uses lie on the same path of control flow, then both are equally valid determinants of the definition type.

There are however two possibilities that further complicate the situation. If there exists a path from a variable definition to an exit node of the program, no deduction of the definition type is possible from uses of the variable occuring past that node. This is because the choice as to whether to exit the program or enter one of the successor nodes may be dependent on the type of the variable.

Similarly, if there is a path to a redefinition of the variable, any uses occuring on that path past the redefinition cannot be used for type determination since the path to the redefinition may be taken to redefine the variable to make it compatible with the use. An illustration of this situation is the following graph:




Here the use of $x$ in block 3 cannot be used to determine the type of the definition in block 1.

We shall now give a formal definition of variable type deduction by method (ii) in terms of a series of equations which use the following notational conventions. Gives types $t_1$ and $t_2$, the operation $t_1$ <u>*alt*</u> $t_2$ produces a type $t_3$ which indicates that the object under consideration is of type either $t_1$ or $t_2$ . Similarly, the operation $t_1$ <u>both</u> $t_2$ produces a type which indicates that the object is of both type $t_1$ and $t_2$, if such a type is possible. If $b$ is a block, we indicate its entry by $y_b$ and its exit by $e_b$. The inverses of these functions are written $b_y$ and $b_e$ respectively. In what follows, *tg* indicates the general type about which nothing is known, *backtype* is the function which determines the type from the way a variable is used in a given use and *du* is a function which, given a definition and a block, returns all possible uses of the defined variable in that block.

The equations which follow are for a function *tfu* defined on block entrances and exits and which give the type deduced from use for an definition, *def:*

(1) tfu(e) = tg if $e$ is a program exit or if $b_e$ contains a definition other than *def* of the variable defined by *def*.

(2) tfu(e) = [<u>alt</u>: pεcesor($b_e$)] tfu($y_p$) otherwise

(3) tfu(y) = [ <u>both</u> : uεdu(def, $b_y$)] backtype(u) <u>both</u> tfu ($e_{(by)}$),

The final deduced type of *def* will be *tfu*($y_b$), where $b$ is the block containing *def*.

Before taking up the details of the typechecking algorithm itself, it is well to define the representation of a program on which it operates. A <u>program</u> is considered to be five-tuple of the form

< <u>nodes</u>, <u>progrph</u>, <u>entry</u>, <u>cesor</u>, <u>cons</u>, <u>exits</u> > where:

<u>nodes</u> is the set of basic blocks in the program.

<u>Progrph</u> is the program graph and is a mapping which takes each member of <u>nodes</u> into a tuple which represents the operations occuring in that basic block in order of execution.

Each operation is represented by a tuple consisting of the output variable, the operation and the input variables.

entry — is the element of nodes which is the entry point to the program.

cesor — is a mapping from nodes into $2^{\underline{nodes}}$ which gives for each basic block, the set of its sucessors.

cons — is the set of constants occuring in the program

exits — is the set of exit blocks of the program.

Preliminary processing of the program determines the following sets:

defs, — the set of all definitions appearing in the program. For the purposes of our algorithm, a definition is a triple consisting of the defined variable, the basic block within which the definition appears, and the integer which gives the position of the definition within the basic block.

defsreaching, — a function which maps each basic block into the set of all definitions which are "live" at the entry to that block; i.e. such that the use of a variable within the block may refer to the value of that variable given by one of these definitions. This is determined by a use-definition chaining algorithm.

The first step of the typechecker, which may more properly be considered as part of the use-definition chaining algorithm, is to create two mappings: *ud* and *du*. The first associates with each use of a variable the set of all definitions which may determine the value of the variable at that use. The second is the inverse of the first; it associates with each definition. The set of all uses where the value of that definition may be utilized.

For the purpose of this algorithm, a *use* is a quadruple consisting of the variable name, the basic block in which the use appears, the position of the operation within which it appears in the basic block, and the integer which tells which input variable it is within the operation.

The algorithm for calculating *ud* and *du* is as follows:

```
define    udfct;

     / * defsreaching, progrph ud and du are assumed global * /

         ud = nℓ;     du = nℓ;
         ( ∀ block ε progrph) optupℓ = block(2)
         / * optupl is the set of operations in a given block * /
               defset = defsreaching (block(1) is node);
               ( 1 < = ∀i < = #optupℓ) result =(optupℓ(i) is opti)(i);
                      op=opti(2);
                     d = <result, node, i>; /* set up the definition
corresponding to the i th operation in the block */
         (3 < = ∀j < = # opti)  u = <opti(j), node, i,j>;
         / * set up the use corresponding to the jth variable
             used in the ith operation in the block * /
         ud(u) = { x ε defset | x(1) eq u(1)};
         / * the set of definitions which could possibly apply
          to use  u * /
          end ∀j;

     s = {x ε defset | result eq x(1)};
     / * update defset by removing all definitions which define
         the same variable as the current definition, adding the
         current definition * /
         defset = defset - s with d;

     end ∀i;
     end  ∀ block;
     (∀ d ε defs) du(d) = nℓ;;
     (∀ x ε ud , d ε x(2)) du(d) = du(d) with x(i);;
     / * this sets up du as the inverse function of ud * /
end udfct;
```

We now move to a description of the typechecker proper. The result of the typechecker will be a mapping *typ* which assigns to each definition and constant in the program its deduced type. We leave a discussion of how we represent these types for later, but mention that we initialize the types of constants to be their types and the types of all definitions to be the "undefined" type signifying that nothing is known about their types.

We then invoke the routine *grafproc* which is in charge of the global management of the typechecker. *Grafproc* keeps a set of definitions, *work*, which consists of all definitions whose type may be determined. Initially, *works* consists of all the definitions in the program to be processed. A single definition is removed from *work* and its type determined by the routine *defproc*. The determination of the type of a single definition enables us to determine the type of two other groups of definitions.

(i) Those definitions which result from an operation applied to a use of the variable whose type has just been determined.

(ii) The determination of the type of this definition may enable the determination of the type of one of the variables used in this definition, which in turn may enable the determination of the type of the definition where that variable was defined.

Note that the first group corresponds to the first method of typechecking discussed above and the second group to the second method.

The definitions in the first group are given by:

$$[\ +\!:\ u\ \varepsilon du(d)]\ \{\ df\ \varepsilon\ defs\ |\ \underline{t\ell}\ df\ \underline{eq}\ u(2{:}2)\}$$

while those in the second group are given by:

$$[\ +\!:\ u\ \varepsilon\ usepile]\quad ud(u)$$

where *usepile* is a tuple of all the uses which make up the current definition *d*. *Usepile* is a global variable which is built up by *defproc* in processing the definition *d*.

The code for *grafproc* follows:

```
define grafproc;
        / * defs, usepile are global * /
        / * initialize workset * /  work = defs;
        ( while work  ne nℓ)  d  from  work; /* remove a definition
                                                   from work * /
        if defproc(d) then
        work = work + [ +: u ε du(d)] {df ε defs | tℓ df eq u(2:2)}
                      orm  nℓ +[+: u ε usepile] ud(u);;
        end while,
        return;
end grafproc;
```

The routine *defproc* processed the definition which is its argument, determining its type. If this type differs from its original type signifying that more information is known about the definition, *defproc* returns *true* which is a signal to *grafproc* to add all definitions which may be affected by a type change in this one to the set *work*.

We now give the code for the routine *defproc*. This function calls on two other functions *newtyp* and *back* which determine the type of a definition by methods one and two respectively; that is, *newtyp* combines the known types of the variables which make up the definition, utilizing the operation in the tuple to produce the type of the result, while *back* searches for all uses of the defined variable and combines type information deduced by the way the variable is used into a type for the variable. These two returned types for the definition are then combined by the function *both* which produces a "lowest common denominator", i.e. the type of an object known to have both of two given types. If this resultant type is different from the known type of the definition on entry to *defproc*, we return *true*, else *false*. *Defproc* is also responsible for building the tuple *usepile* of all uses which make up the input dedinition.

```
definef  defproc(d);
         / * usepile, progrph, typ are global * /
         usepile = nult; modif = false;  optupℓ = progrph (d(2));
         result = (optupℓ (d(3)) is opt3)(i);
         op = opt3(2);     oldtype = typ(d);
         (3 < = ∀j < = # opt3) usepile (j-2) = <opt3(j),d(2),d(3),j>;;
         typ(d) = both (back(d), newtyp(op, usepile));
         if typ (d) ne oldtype then modif = true;;
         return modif;
end defproc;
```

We now turn to a discussion of representing types and combining them. We distinguish among eight elementary types and represent them by bit string flags having values of powers of 2 (for easy combination) as follows:

*tu* - the type of $\Omega$, the undefined atom

*ti* - integer type

*tb* - boolean or bit-string type

*tc* - character string type

*tn* - null set type

*tt* - null tuple type

*tg* - general type used where the type of an object is too complex for compact representation; can be anything.

*tz* - neutral or error type. Originally before anything is known about the type of an object, its type is *tz*. If its type is still *tz* at the end of processing, we know that an error exists. We assume $tz = 0$.

These elementary types can be combined by *alternation*, that is an object may have type *tin* which signifies that the object is either an integer or the null set. Similarly, we can have any other combination of elementary types. These types are represented as the logical "or" of their constituent types.

Compound types are represented by tuples. The type of a set is a triple $< st, o, type >$ where *st* is an integer flag representing a set and *type* is the type of the elements of the set. Thus a set which contains bit strings and integers would have type $< st, o, tbi >$. A set of integer sets would have type $< st, o, < st, o, ti >>$ which illustrates that types can be nested. To simplify matters, a maximum nesting of 3 is allowed so that the type of a set of sets of sets of sets would be given by:

$$<st, o, <st, o, <st\ o, tg>>>$$

The type of a tuple of unknown length is a triple $<unt, o, type>$, where *unt* is an integer flag representing a tuple of unknown length and *type* is the type of the elements of the tuple. Thus a tuple of unknown length consisting of character strings and null sets would have type $< unt, o, tcn >$.

A tuple which is known at compile time to have length $n$ is represented by a (n+2)-tuple of the form:

$$<knt, 0, type1, type2, \ldots, typen>$$

where *knt* is an integer flag representing a tuple of known length and *typej* is the type of the $j$th component for $1 \le j \le n$. Thus a tuple of length three consisting of a set of integers, an integer, and either the null set or null tuple would be represented as:

$$<knt, 0, <st, 0, ti>, ti, tnt>.$$

The second component of compound types is reserved for indicating alternation with elementary types. For example if an object is known to either be an integer or a set of bit strings, its type would be:

$$<st, ti, tb>.$$

Alternation between two compound types of different grosstype (the *grosstype* of a compound type is its compound type flag, either *st*, *unt*, or *knt*) produces *tg*, the general type.

There are two basic routines for combining types. Given two types *a* and *b*, the function *alt* returns the type of an object which is known to be either of type *a* or of type *b*: This routine assumes that *st* = 2, *unt* = 3, *knt* = 4 and a function *grostyp* which returns the grosstype of a type, defined by: definef grostyp(a); return if integer a then eℓ else a(i); end grostyp; Here *eℓ* is a flag representing an elementary type and it is assumed that *eℓ* = 1; under these assumptions the code for *alt* follows:

```
definef aℓt(a,b); /* first rearrange the types such that
                                    grosstype(a)≥ grosstype(b) */
    if (grostype(a) is ga) gt(grostype(b) is gb) then return
                                                        alt(b,a);;
    /* alternation of non-null set and non-null tuple is tg  */
    if ( { <eℓ,eℓ,eℓeℓ>, <eℓ, st,eℓcmp>,<eℓ,unt,eℓcmp>,<eℓ,knt,eℓcmp>,
    <st,st,stst>,<unt,unt,unun>,<unt,knt,unkn>,<knt,knt,knkn>}
    (ga,gb) is label) eq Ω then return tg;;
  go to label;
eℓeℓ: /* both operands elementary */
    return if a eq tg or b eq tg then tg else a + b;
eℓcmp: /* elementary type and compound type */
    return if a eq tg then tg else if a eq tz then b
         else <b(1), a+b(2)> + tℓ tℓ  b;
stst: unun: /* two sets or two tuples of unknown length */
    return <a(1), a(2)+b(2), alt(a(3), b(3))>;
unkn:  /* a tuple of unknown length and one of known length */
    return <unt, a(2)+b(2), alt (a(3), [alt:3<=i<=#b] b(i))>;
knkn:  /* two tuples of known length, if both have same length then
   the result is a known tuple, consisting of the alternation
      of corresponding individual elements, otherwise an unknown
    tuple consisting of the alternation of all elements */
```

```
     return if (# a is na) eq (#b is nb) then
               <knt, alt(a(2), b(2))>  + [+:3<=i<=na]<alt(a(i),b(i))>
            else <unt,alt(a(2),b(2)),alt ([alt:3<=i<=na]a(i),
                                          [alt:3<=i<=nb]b(i) )>;
end alt;
```

Here, alt is an operator defined by:

```
definef  a alt b; return  alt(a,b); end;
```

Similarly, the routine *both* receives two types as input parameters and returns the type of an object which is known to be of both types. The code follows:

```
     definef both (a,b);
          /* if either a or b is  of general type, return the other */
          if a  eq tg then return b;;  if b eq tg then return  a;;
          /* s is a flag which is zero if and only if one of a   and
          b  are elementary */
          s=0; if atom  a then ja=a;  else ja=a(2); s=1;;
               if atom  b then jb=b; s=0; else jb=b(2);;
          /* ja and  jb  are the elementary parts of  a and b */
          if s eq 0 then return ja*jb;;

kntup:    /* here we test for one of the types being a tuple of
               known length */
          if grostyp(a) eq knt then tup=<knt,ja*jb>;
               if grostyp(b) eq unt then
                    (  2<∀i<=#a) tup(i)=both (a(i),b(3) );;
                     return tup;
               end if;
               if grostyp(b) eq knt then
                   if (#a) ne (#b) then return ja*jb;;
                   (   2<∀i<=#a) tup(i)= both (a(i), b(i) );;
                   return tup;
               end if;
```

```
        end if; /* if b is a known length tuple, swtich a and b */
        if grostyp(b) eq knt then  c=b; b=a; a=c; goto kntup;;
    if a(1) eq b(1) then return
        if (both (a(3), b(3) is  bo)eq 0 then ja*jb
            else <a(1), ja*jb, bo>;;
        return ja*jb;
end both;
```

We now turn to descriptions of the actual processes of deducing the type of a definition. The routine *newtyp* is responsible for combining the types of the uses which make up a definition according to the operation to determine the type of the result (method (i) above). Its input parameters are *op* which is the operation and *u* which is a tuple of the uses making up the definition (this tuple was prepared by *defproc* which calls *newtyp*). *Newtyp* divides all operations into several categories. Some operations (like division or equality testing) preordain the type of their result without regard to the types of their inputs. Others (like plus or assignment) do depend on their inputs and can be divided into binary , unary or special operators. For binary operations, *newtyp* divides the input types into their compound and elementary parts, determining the result types for each combination of parts, and then taking the alternation of the whole.

The opcodes which the algorithm presently handles are the following:

*odv* - integer division
*oabs*- absolute value
*ohd* - head of a tuple
*otℓ* - tail of a tuple
*oarb* -arbitary element of a set
*oass* - assignment
*opw* - power set of a set
*odec* - decimal converter
*ooct* - octal converter
*osiz* - number of elements,bits or characters
*onot* - logical negation
*oad*- plus,unron, concatenation
*osb* - minus, set difference
*omℓ* - multiplication, intersection, replication
*orm* - remainder, symmetric difference

*omxm* - maximum
*omnm* - minimum
*oeq,one,olt ole, ogt, oge* - comparison
*oand, oor* - logical operations
*oelm* - element test
*owith* - SETL with operation
*olss* - SETL less operation
*olsf* - less functional values
*oinc* - inclusion test
*oof* - function application, position extraction
*oofa* - multivalued function application
*onpw* - SETL npor operation
*ondx* - indexing of tuple, bit or character string
*oset, otpl*- set or tuple former
*ondxass* - indexed assignment
*ord* - read operation

An operation is represented by a tuple; the first element is the variable into which the result is stored, the second is the opcode, and the others are the operands. Thus the operation x+y would be represented by <'*ti*', *oad*,'*x*', '*y*'> where *ti* represents a temporary location. An exception is made in the case of indexed assignment where the indexed quantity is included among the operands (this is because the previous type of the quantity enters into the determination of the eventual type). For example a(x)=q would be represented by < 'a', ondxass, 'x', 'q', 'a' >. Note that the indexed quantity appears as the last member of the tuple, and the value to be assigned as the next to last.

The code for *newtyp* follows:

```
definef newtyp(op,u);
     /* op is the operation, u the tuple of the operands. */
            if op eq ord then return tg;;
     if op ε {odv, oabs, omxm, omnm,odec,ooct,osiz} then  return ti;;
     if op ε { oeq,one,alt,ogt,oℓe,oge,oand,oor,onot,oelm, oinc}
                                                then return tb;;
```

```
     spoplab = {<oass, asscas>,<oset, setcas>, <otpl, tplcas>
                <ondxass, xacas>}; /* function which returns a label */

     nu=#u;
     argl=u(l); targl=argtyp(argl);
     go to
        if op ε { oass, ondxass,o set,otpℓ} then spoplab(op)
     else if op ε { ohd,otℓ, oarb,opw} then unop
     else if op ε { oad,osb,omℓ,orm,owth,oℓss,oℓsf,oofa,onpw, oof}
          then binop  else pℓop;

asscas: return targl;
setcas: return <st,tz,[alt: l< = i < = nu] nstchk(argtyp(u(i)))>;
tplcas: return <knt,tz>+[+: l< = i < = nu]<nstchk(argtyp(u(i)))>;
xacas:  typset = nℓ; /* typset is an accumulator set for
                        all the alternatives that the
                        result type could be */

        if nu eq 3 and both(targl, ti) eq ti then
        /* in this case the indexed assignment may be either to
        a set or a tuple (note that indices of more than one
        integer into tuples of tuples are not covered) */
          if (argtyp(u(#u)) is targu) eg tg then return tg;;
          if both (targu,tb) eq tb then tb in typset;;
          if both (targu,tc) eq tc then tc in typset;;
          if both (targu,tt) eq tt then
              if both (argtyp(u(2) is targ2,tu)eq tu then tt in typset;;
              if both (targ2, tg) ne tu then <unt,0,targ2>in typset;;
        end if;
              if grostyp(targu) ge unt then
              /* here we assume the flag for st is 2,
                 that for unt is 3, and for knt is 4 */
                 <unt, 0,alt(argtyp(u(2)), [alt:2<i<=#targu] targu(i))>
                                                      in typset;;
       end if;
```

```
      /* we are now at the possibility of indexing into a set */
      targ2 = newtyp(otpℓ, u(1:(#u)-1));
      cmcmbin(owth, argtyp(u(#u)), targ2) in typset;
      /* treat it as the set with the tuple which may be inserted */
      return [alt: i ε typset] i;
 unop: return if grostyp(targ1) eq eℓ the eℓun(op,targ1)
                else elun(op, targ1(2)) alt cmpun (op, targ1);
      /* elun takes care of finding the type for a unary
         operator on an elementary operand and cmpun on the
         compound part of the operand */
 binop: targ2= argtyp (u(2)); binℓab= {<eℓ,eℓ,bineℓeℓ>,<eℓ,st,bineℓst>
                                      <st,eℓ,binsteℓ> <st,st,binstst>};
      go to binℓab(grostyp(targ1) min st, groseyp(targ2) min st);
 bineℓeℓ: return eℓeℓbin(op,targ1,targ2);
 bineℓst: return eℓeℓbin(op,targ1,targ(2)) alt eℓcmbin(op,targ1,targ2);
 binsteℓ: return eℓeℓbin(op,targ1(2),targ2) alt cmeℓbin(op,targ1,targ2);
 binstst: return eℓeℓbin(op, targ1(2),targ2(2))alt eℓcmbin(op,targ1(2),
                targ2) alt cmeℓbin(op,targ1,targ2(2))alt cmcmbin
                                                       (op,targ1,targ2);

      /* these four routines (elcmbin, elelbin,cmelbin, cmcmbin)
         combine types depending on their grosstype */
plop:   arglst=tℓ u; /* the only plunary operator currently is index
                           of a tuple or string */
      return if grostyp (targ1 eq eℓ then eℓpℓu(op,targ1,arglst)
           else eℓpℓu(op,targ1(2),arglst) alt cmppℓu(op,targ1,arglst);
 end newtyp;
```

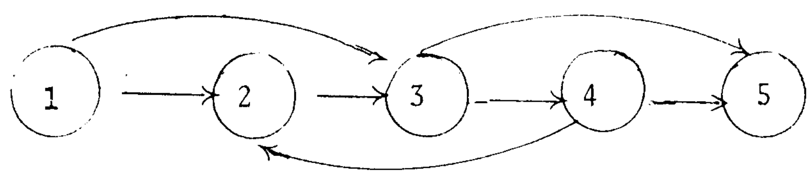
Aside from the type-finding routines, *newtyp* uses two auxiliary routines which have not yet been discussed. *argtyp* returns the type of its argument. The code follows:

```
 definef argtyp(arg); /* arg is a use of a constant or variable */
       /* cons, typ, and  ud  are global */
       if arg(i) ε cons then return typ(arg(i));;
          return [ alt: x ε ud(arg)] typ(x);
 end argtyp;
```

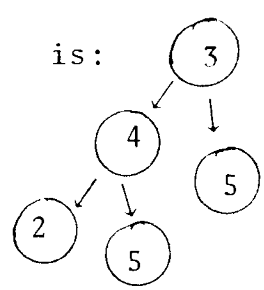
The other routine *nstchk* insures that nesting is never deeper than three by checking something which is being nested for double nesting

```
definef nstchk(x);
     if grostyp(x) eq eℓ then return x;
     ( 2 <∀i<=#x)  if grostyp (x(i)  is xi) ne eℓ then
              ( 2 < ∀j<=#xi)  if grostyp (xi(j)) ne eℓ then xi(j)=tg;
                                                          x(i)=xi;;
               end ∀j; end if;
    end ∀i;
    return x; end nstchk;
```

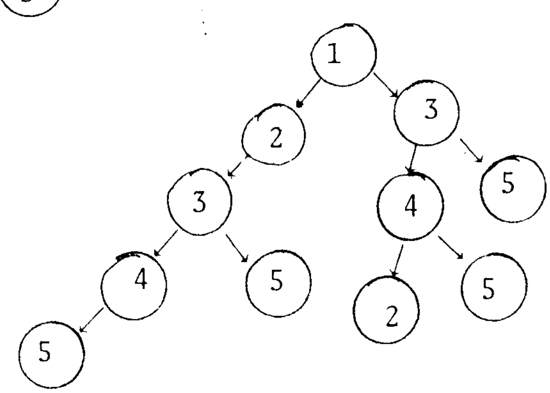
We now rurn to a description of typechecking by method (ii). In order to solve the equations (1) - (3) presented earlier, we introduce the concept of the <u>tree of a program rooted at a given node</u>. This tree is constructed from the program graph by establishing the given node as root and using the graph successors as the tree successor so long as they do not cause a cycle. For example, given the program graph




the tree rooted at node   is:




the tree rooted at node 1 is:



we then assume type *tg* at the exits of the leaves and travel up the tree propagating types according to the aforementioned equations.

Given a definition *d*, the function *back* is responsible for constructing the tree and then calling a function to walk the tree and determine the type of *d* by method ii.

The code for the routine *back* is as follows:

```
definef back(d);
     t = progtree(d(2));    /* build the tree */
     return typfind (d,t); /* determine the type from the tree */
end back;
```

The code to build the tree is:

```
definef progtree (node);
     /* succ is the global set of tree successors, cont a global
        map assigning to each tree node, the program node which
        it represents, tpred a local map giving all nodes in the
        path from the tree root to any given node, cesor, the
        global graph successor */
        succ = nℓ; cont = nℓ; tpred = nℓ; t = newat;
        cont(t)=node;  work={t};  tpred(t)={t};
        (while work ne nℓ) tnode from work;
              succ(tnode) = nℓ;
              (∀cε cesor (cont(tnode)))  if c n ε cont[tpred(tnode)]
                   then b = newat; cont(b)=c;
                   tpred(b)=tpred(tnode) with b;
                                 succ(tnode)=succ (tnode)with b;
                   b  in  work;  end if;
            end ∀c;
      end while;
      return t;
end progtree;
```

The routine *typfind* is responsible for walking down the tree and determining the types of the uses and how they combine in enabling us to deduce the type of the defined variable. Note that two cases must be specifically checked for as discussed earlier. If a node along the tree is an exit or it contains a redefinition of the variable, the tree traversal need not go beyond that node. Note further that if a redefinition occurs within a node, it will occur after any uses of the previous definition; since if it occurred before, the uses would be uses of the redefinition and not of the original. Thus all uses within the block where redefinition occurs are valid determinants of the original definition type.

The code for *typfind* follows:

```
definef  typfind (d,t);
     /* cont, succ, du, exits, defs are global */
        j = [both: uε du(d) | u(2) eq cont(t)]
                        backtype(oper(u), u(4), restyp(u) orm tg);
        s = succ(t);
        if s eq nℓ or cont(t) ε exits or ∃ df e defs |(df ne d
                and d(1) eq df(1) and df(2) eq cont(t) )
                    then return  j;;
        return both (j, [ alt: trεs] typfind(d,tr) );
end typfind;
```

This routine uses the following auxiliary routines:
a) *backtype* which does the actual determination of type of a use depending on the following parameters: the operator, the position within the operation tuple of the use, and the type of the operation result (which may have been determined on the basis of other uses with known types). The actual code for *backtype* is a detailed accounting of all possibilities and will not be given here.

b) *oper* which determines the operator which is applied to a given use. The code is:

```
definef oper(u); return (( progrph(u(2)) ) (u(3)) ) (2); end oper;
```

c) *restyp* which gives the type of the result of an operation in which a given use occurs. The code is:

```
definef restyp(u); return type(<((progrph(u(2)))(u(3)))(1),u(2),u(3)>);
                                                           end restyp;
```