J.T. Schwartz
July 25, 1974

SETL Newsletter # 137

# Additional Thoughts Concerning Automatic Data Structure Choice

## 1. Introduction. The Subpart Graph of A Program.

By exploring designs leading to efficient implementations of various of the abstract algorithms presented in O.P. II, one can make oneself aware of the wide range of structural facts which are exploited when data structures are chosen manually. Some of the facts exploited are of a type which can be discovered in reasonably straightforward ways and recur often: these are obvious candidates for incorportion into an optimiser. Other facts are deep or of rare occurence; these should be regarded as mathematical transformations which can probably not be encompassed by a reasonable optimisation algorithm at the present time. Between these two extremes lies a range of marginal facts: their consideration suggest interesting possibilities for automatic analysis, but possibilities whose actual profitability remains subject to question. In the present newsletter we will discuss in interrelated group of techniques having this marginal character; later investigations may show how these techniques or variants of them can be made practical.

We begin by describing a technique which in some cases will allow us to describe the types of the objects appearing in a SETL program P more precisely than would be possible if only Tenenbaum's typefinder were used. This is done as follows: We form a directed graph G called the *subpart graph* of P. The nodes of this graph are the ovariables of P (for notation and terminology used in this newsletter, see NL 130 and 131).

The edges of the are defined by the value-flow functions *crthis*, *crmemb*, *crcomp*, and *crsomcomp* introduced in NL 131. We draw an *m-edge* from o' to o if o'∈ [+: i∈ crmemb(o)] crthis(i); that is, if o' can create a value which at some point in the execution of P becomes a member of the value of o. Similarly, we draw an *s-edge* from o' to o if o' can create a value which at some point in the execution of P becomes a component(unknown position) of the value of o, and an (c,n)-edge if o' can create a value which at some point in the execution of P becomes the n-th component of the value of o. These last two conditions are equivalent to o'∈ [+: i∈somcomp(o)] crthis(i) and o'∈ [+: i∈ comp(o,n)] crthis(i) respectively. As an illustration of this graph, consider the short program

(1) $s = \underline{n\ell}$; $(1 \le \forall i \le 100)$ $s = s + \{\{s\}\}$;;

To show all the ovariables in this program, we expand it and mark seperate ovariable occurences of s and i with distinct subscripts. This gives

(2)

$s_1 = \underline{n\ell}$;

$i_1 = 1$;

loop: $x = \{s\}$;

$y = \{x\}$;

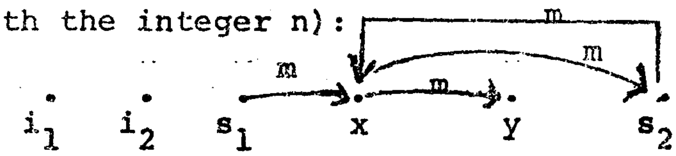
$s_2 = s + y$;

$i_2 = i + 1$;

if i $\underline{\ell t}$ 100 then go to loop;;

The subpart graph for this program may be diagrammed as follows (we mark m-edges with an m; s-edges with an s; and (c,n)-edges with the integer n):

(3)




This graph contains a loop; a fact which clearly reflects the recursive nature of the data structures built up by the program (1) (and by its equivalent(2)).

To analyse the data objects generated by a program whose subpart graph G contains a loop, we proceed as follows: Choose a minimal set of K of nodes in K. Each node in K is then regarded as the name of some (recursive) object type. Type descriptors are assigned to the remaining ovariables of G as follows: The graph G-K has no loops, and consequently can be sorted topologically. Sort it, and take its ovariables in decreasing order. The type symbol τ assigned to an o∈G-K is calculated as an alternation $\tau = \tau_1 | \tau_2 | \ldots | \tau_n$ of type symbols $\tau_j$ which are individually determined in the following way:

i. If o has an incoming m - edge from a node o' to which the type symbol a has been assigned, take $\tau_j = \{a\}$ as an alternand;

ii. If o has an incoming (c,n)-edge from a node o' to which the type symbol a has been assigned, then the values of o will be tuples of some known length ℓ. In this case, take $\tau_j = \langle tz,\ldots,a,\ldots,tz\rangle$ as an alternand. Here tz is the 'error' of 'overspecific' type which functions as the unity element for alternation in the calculus of types; and the a occur in the n-th of ℓ component positions. If in addition o has an incoming s-edge from a node o" to which the type symbol b has been assigned, then take $\tau_{j'} = \langle b,b,\ldots,b\rangle$ as an alternand.

iii. If the immediately preceeding rule does not apply, but if o has an incoming s-edge from o', take $\tau_j = [a]$ as an alternand.

Once these rules have been applied to all the nodes in G-K, they can be applied to the nodes K, and when so applied will generate recursive descriptions of the type symbols originally generated for the nodes of k. For example, if in connection with the graph (3) we take K = $\{s_2\}$, then the type description

(4)
$$\begin{aligned} s_1 &\sim \underline{n\ell} \\ x &\sim \{\underline{n\ell} | s_2\} \\ y &\sim \{x\} \\ s_2 &\sim \{\{\underline{n\ell} | s_2\}\} \end{aligned}$$

results. Note that we can always disrupt the cycles of G by removing a set K of programmer defined (rather than compiler specified) ovariables; and will generally prefer to do so.

The subpart graph of a program P represents certain coarse aspects of the data structures built up by P. It is possible that this graph can be used to guide the activity of an automatic program analyser. At the very least, the type description generated by use of the subpart graph of a program will retain some of the information lost in Tenenbaum's typefinding algorithm because of the restricted way in which nested types are handled in that algorithm.

## 2. An example. Basic sets and an automatic method for introducing them. 'Permanently' and 'temporarily' applied operators. 'Harmless' Transformation of data structures. Cases in which a basic set becomes dead.

As an additional example illustrating certain interesting issues which arise in data-structure choice, consider the interval-finder routines given on pp. 269-272 of O.P.II; which routines we now repeat (with some rather small modifications) for the readers convenience.

```
definef interval(nodes,x);
/* npreds, followers and cesor are assumed to be global */
/* count the number of predcessors of every node*/
npreds = {<x,0>, x ∈ nodes};
(∀x ∈ nodes, y ∈ cesor(x))
   npreds(y) = npreds(y) + 1;;
int = nult; followers = {x}; count = {<y,0>,y ∈ nodes};
count(x) = npreds(x);
/* 'count' will be a count of the number of predecessors of
    a node which belong to the interval being constructed */
(while {y ∈ followers|npreds(y) eq count(y)} is newin ne nℓ)
   (∀z ∈ newin)
         int(#int+1) = z;
         z out followers;
         (∀y ∈ cesor(z)|y ne x) count(y) = count(y)+1; y in followers;
   end ∀z;
end while;
return int;
end interval;
```

```
definef intervals(nodes,entry);
/* followers, follow, intov are all assumed to be global */
ints = nℓ; seen = {entry}; follow = nℓ; intov = nℓ;
(while seen ne nℓ)
    node from seen;
    interval(nodes,node) is i in ints;
    follow(i) = followers;
    (1 ≤ ∀k ≤ #i) intov(i(k)) = i;;
    seen = seen + followers;
end while;
    return ints;
    end intervals;

definef dg(nodes,entry);
    /* cesor, follow, intov, dent, pred are all assumed to be global*/
ints = intervals(nodes,entry); dent = intov(entry);
(∀i ∈ ints) cesor(i) = intov [follow(i)];;
return ints;
end dg;

definef dseq(nodes,entry, graphcesor); /*dent and cesor are global*/
seq = <<nodes,entry>>; <n,e> = <nodes,entry>; cesor=graphcesor;
(while #(dg(n,e) is der) lt #n doing <n,e> = <der,dent>;)
     seq(#seq+1) = <der,dent>;;
return seq;
end dseq;
```

One of the things at which an efficient implementation of the interval-finding code shown above will aim is the representation of the maps *follow*, *intov*, and *cesor*. Efficient automatic representation of a map f with domains of complex structure is likely to depend on the existence of a *basic* set for f, i.e., of a set s such that $f \subseteq :_1 s$ (in the notation of NL 130 which we now begin to use heavily).

In the interval-finding code there appears no set which includes the domains of all three maps *follow*, *intov*, and *cesor*. However, one can easily be introduced: we have only to insert the instruction *allnodes* = *nodes* after the first line of the routine *dg*, and insert *i* in *allnodes* after the line interval(nodes,node) is i in ints (which is line 6) of the routine *intervals*. Once this set is introduced, the following relationships will be found:

follow $\subseteq_1$ allnodes; follow $\ni 2 \ni$ allnodes; intov $\subseteq_1$ allnodes;
intov $\subseteq_2$ allnodes; cesor $\subseteq_1$ allnodes; cesor $\ni 2 \ni$ allnodes;

In addition, the following relationships are found for subsidiary variables occuring in the interval finder:

npreds $\subseteq_1$ allnodes; followers $\subseteq$ allnodes; count $\subseteq_1$ allnodes;
newin $\subseteq$ allnodes; x $\in$ allnodes; y $\in$ allnodes; z $\in$ allnodes; etc.

The set *allnodes* might be generated automatically in the following way: if we ignore destructive object uses (cf.NL 131, section 2) and examine the subparts graph of our program then we see that elements belonging to the domain of *follow*, *intov*, *cesor* (and also *npreds* and *count*) are generated in only two ways: from the parameter *nodes* when the principal routine *dg* is entered, and by the instruction *int* = nult appearing in the seventh line of the routine *interval*. This suggests the utility of forming a set into which all the elements of the *nodes* parameter, as well as all the objects generated by the instruction *int* = nult will be placed. However, an instruction placing an object x in a basic set should be placed along a minimal collection of paths which seperate the destructive uses of x from the operations which make x part of an explicit program object.

For the code shown above, this means that *int* should be inserted into *allnodes* after exit from the *while* loop of *interval* and before the statement (5) of the routine *intervals* which is just what we have proposed.

It is fairly typical for the objects in SETL program to be generated, used destructively at first, and used non-destructively thereafter. This usage pattern is seen for several of the objects appearing in the above code; in particular, for *followers* and *ints*, and for *cesor* in the larger optimisation context of which the interval-finding routines shown above from a part. Given an object x generated at an ovariable o, call an appearance of x at a point not leading to any destructive use of x an appearance of x in *permanent form*, and call an appearance of x at a point leading to a destructive use of x an appearance of x in *temporary form*. Operations to which a permanent form of x is an argument we call operations *permanently applied* to x; if either a permanent or temporary form of x is an argument of an operator op, we say that op is an operator *applied* to x. For this purpose, iteration is reckoned as an operator, which is by definition applied to any object which in one of its appearances supports iteration. Objects must be maintained in forms which effectively support all the operations applied to them, but one can distinguish between operations permanently applied and the larger class of operations applied either permanently or temporarily. If this distinction is made, then two seperate object representations can be maintained at all program points leading to destructive uses of an object, and one of these can be thrown away when destructive use becomes impossible. Operator applications in the interval-finding code shown above are as follows:

The object *int* is a vector, with which a precalculated hash should be associated; *seen* can be represented by a list. Since *followers*, *ints*, *seen*, and *temporary* must support insertion operations, four bits should be reserved in each of the elements of *allnodes* to indicate membership/non-membership in these three sets.

A basis set like *allnodes*, which is used only as a means for representing other sets and maps, will only be consulted explicitly if one of its elements is a composite whose inner details need to be retrieved, or if an object not directly represented in terms of the basic set must be combined with a set or map represented in terms of the basic set, or if an object which may be new (but need not be new) needs to be added to the basis set. Once one has reached a program location at which all such operations have become impossible, the basis set is useless and can be dropped. This remark applied to *allnodes*, which becomes useless and can be suppressed on return from the routine *dseq*.

In some cases, the remark that has just been made will apply to tell us that a particular basis set need not be generated at all. Such cases, the basis set serves merely as a conceptual device; essentially it imposes an encoding on its nominal members, converting them from explicitly represented objects, and making them pointers or integers.

A suitably powerful program-analysis/structure-choice algorithm should be capable of generating the data-structure design outlined in the paragraph following table I. The quality of this design begins to approach that of good manually developed design. However, a good manual design for the algorithm we have been considering will exploit a few special observations which allow some signficant savings to be made.

(note that we include one compiler-generated *temporary* variable in the list of variables appearing in our table; this is the *temporary* which stores the value intov [follow(i)] calculated in the fourth line of the routine *dg*. This *temporary* plays an especially important role in our analysis; other temporaries, whose role is less significant, we elide in order to avoid excessive detail.)

**Table I.**

| object | permanently applied operators | other operations applied |
|---|---|---|
| npreds | indexing | indexed assignment |
| nodes | iteration | |
| cesor | indexing | indexed assignment |
| int | iteration, map application | concatenation |
| followers | iteration, union (nondestructive) | insertion, deletion |
| count | | indexing, indexed assignment |
| ints | iteration, map application | insertion |
| seen | | selection, deletion, union (destructive) |
| follow | | indexed assignment, indexing |
| intov | indexing | indexed assignment |
| seq | indexing | concatenation |
| *temporary* | iteration | insertion |

The information shown in this table might lead an algorithm to the following choice of data structures: Each element of *allnodes* can be treated as an pointer. The maps *npreds*, *count*, *cesor*, *follow* and *intov* can be kept in fields attached to the elements of *allnodes*. The permanent form of *nodes*, *followers*, *ints*, and *temporary* can be lists; *followers* should be a two-way list because deletions are applied to it during its construction.

The object *int* is a vector, with which a precalculated hash should be associated; *seen* can be represented by a list. Since *followers*, *ints*, *seen*, and *temporary* must support insertion operations, four bits should be reserved in each of the elements of *allnodes* to indicate membership/non-membership in these three sets.

A basis set like *allnodes*, which is used only as a means for representing other sets and maps, will only be consulted explicitly if one of its elements is a composite whose inner details need to be retrieved, or if an object not directly represented in terms of the basic set must be combined with a set or map represented in terms of the basic set, or if an object which may be new (but need not be new) needs to be added to the basis set. Once one has reached a program location at which all such operations have become impossible, the basis set is useless and can be dropped. This remark applied to *allnodes*, which becomes useless and can be suppressed on return from the routine *dseq*.

In some cases, the remark that has just been made will apply to tell us that a particular basis set need not be generated at all. Such cases, the basis set serves merely as a conceptual device; essentially it imposes an encoding on its nominal members, converting them from explicitly represented objects, and making them pointers or integers.

A suitably powerful program-analysis/structure-choice algorithm should be capable of generating the data-structure design outlined in the paragraph following table I. The quality of this design begins to approach that of good manually developed design. However, a good manual design for the algorithm we have been considering will exploit a few special observations which allow some signficant savings to be made.

Specifically: all the members of the *nodes* parameter of *deeq* are apt to be atoms; and if they are no node will be inserted more than once into the set *followers* of the routine *interval*; and thus no node or interval will be generated more than once. It is therefore totally unnecessary to maintain the set *allnodes*; one need only generate a serial number for each element which would be added to this set if it were maintained. The maps *npreds* and *count* are only defined on the subset *nodes* of *allnodes*; and the map *followers* is only defined on the subset *ints*. Moreover, if the elements of *allnodes* are numbered in their order of generation, *nodes* and *ints* are always contiguous collections of integers within *allnodes*. Thus these sets need not be maintained as lists, but can be represented simply by a pair of integers, one representing a first set member in serial order, the other a last set member. The maps *npreds* and *count* can be represented by vectors v of integers,the i-th component of these vectors representing the value of *npreds(x)* or *count(x)* i-th smallest member of *nodes*. Much the same device can be applied to the representation of *follows*. These concrete design improvements should reduce by a factor of approximately 4 the amount of storage space required by an implemented version of the interval finder algorithm; speed should increase only slightly above that attained by use of the automatically chosen data structures described a few paragraphs above.

It is worth noting that all of the concrete design improvements presented in the preceeding paragraph rest on the single fact that no interval is ever generated twice by the routine *interval*. This fact is probably of too great a logical depth to be uncovered by presently available automatic analysis techniques.

However, it is easily *surmised*, and in a suitably interactive system an inquiry as to its truth might be generated. Once this fact becomes known, it is easily seen that the members of *ints* must form a continuous range within *allnodes*; indeed, *ints* is initialised to nl, no deletions are ever made from it, and members of *allnodes*, once generated, are inevitably inserted into *ints*. Since *nodes* is always set from *ints*, *nodes* has this same property.

If an optimiser system undertakes to choose data structures automatically, it is important that it report its choices in some comprehensible form to the system user. It is also important that circumstances preventing substantially superior choices from being made should be reported. This can make the user aware of code details which he may regard as innocent and could easily change but to which the data structure choice mechanism reacts strongly. Then, by modifying these details, he may be able to obtain a substantially better implementation.