SETL Newsletter 131 June 7, 1974

More on Copy Optimization of SETL Programs J. Schwartz

1. Introduction.

SETL is by definition a "value" rather than a pointer language. This can force SETL objects to be copied considerably more often than one would like (see NL 57). It is therefore important to find ways to reduce the number of copy operations that appear in the compiled from of a SETL program P; indeed, this may be the single most important optimization that can be applied to P. The present newsletter will describe a method for analyzing SETL programs to find contexts in which copying can be omitted. This method, which is purely static, can also be applied in connection with the partly value, partly pointer semantics of the present SETLB system to detect possible deviations from strict value semantics and issue appropriate warnings. An analysis of the kind we envisage might also be capable of producing information which makes it easier for a SETLB user to improve his programs by inserting explicit copy operations at a few crucial points.

A program P to be analyzed is assumed to be schematized in the manner described in NL 130, i.e. to be represented as a collection of basic blocks, each block consisting of schematized instructions which we will write either as

$$v = op(i_1, \ldots, i_n) \tag{1}$$

or in infix or in any other convenient form. As in NL 130, we shall refer to a specific occurrence of a variable v in 'target' position within an instruction (1) as an *ovariable*, and to a specific occurrence of a variable i_j in argument position within an operation (1) as an *ivariable*. An ovariable will sometimes be taken to stand for the operation (1) which sets its value.

'Long' SETL objects are always accessed via 'root words' or 'root pointers' which point to the memory block in which the actual representation is stored. If to simplify things we abstract from the action of SETL's garbage collector and assume an infinite amount of memory to be available, then these pointers can always be taken to contain unique adresses, and to be in 1-1 correspondence with SETL 'values'. It is then well to think of each pointer as pointing to its own private 'segment', infinite in length, within which an indefinitely large data item may be built up. When a 'long' SETL object i is made part of a SETL composite c (either a set or a tuple) we very much prefer to insert i's 'root pointer' rp into c without copying; the same remark applies to simple assignments which transfer the value of one variable to another. If this is done, then rp may be extracted later, and used; if the rules of value semantics are to be adhered to, we must be sure that rp still points to a memory block representing the value i. The problem we face is to harmonize this requirement with our desire to improve efficiency whenever possible by modifying existing data objects rather than by continually creating new objects which differ only slightly from old ones. This dilemma appears whenever we meet the instruction

(2) s = s with x;

especially when this instruction appears within a loop. The instruction (2) can be implemented in one of two ways: either by creating a new copy *nbod* of the body *bod* of s (and correspondingly a new pointer to this data object); or by modifying the body of s (by inserting x into this body), in which case the old pointer p now points to the representation of s with x and no new pointer is created. If *bod* is large, the second procedure can of course be much more efficient than the first. However the second procedure may be said to be *destructive* of the pointer p, in that it destroys the

relationship between p and the SETL value which it formerly represented. We must guard against destructive pointer uses which have unanticipated side-effects; to do so, we must trace the direct and indirect flow of pointers through P, from the points at which a new memory block is allocated and a new pointer p is created, out to all the objects which come to incorporate p, and out to the places at which we finally decide to use p destructively.

The central idea of the scheme to be developed may be put as follows. When a pointer p becomes the value of an ivariable i which is used destructively by an instruction op of a program P, any object into which p may have been incorporated might undergo unanticipated side effects. We therefore aim to determine the set $\ell(i)$ of all ovariables o of P whose current values might possibly incorporate p; destructive use of p is possible only at program points at which all these o are dead. We find $\ell(i)$ (or rather an upper bound for it) in two steps. First we determine the set *exsinthis(i)* of all instructions which can have been executed between the time at which the pointer p was created and the time p became the value of i. Then we determine the set of all ovariables into whose value one of these instructions may have inserted p; this gives us (an upper bound for) $\ell(i)$.

The technique to be used assumes that P has been subjected to a preliminary 'data flow analysis' or 'definition chaining' which for each ovariable o supplies the set du(o) of all variables which can use the value stored by o, and for each variable i supplies the set ud(i) of all ovariables which can store the value used by i.

The operations for which we may wish to use one argument destructively are s with x , s less x , s lesf x , s(y) = x where s is a set, s(y) = x where s is a tuple,

$s(y_1,\ldots,y_n) = x$, $s\{y\} = x$, $s\{y_1,\ldots,y_n\} = x$, $s + t$, and $s - t$. However, rather than confronting this full range of possibilities immediately, we simplify our discussion (especially in Section 2 below) by ignoring tuple operations completely, and by assuming that the only four set-theoretic operations which appear in our schematized programs are s+t, s-t, {x}, and #s. Note that this does not actually restrict the generality of the set-theoretic constructions we are able to handle, as the other set-theoretic operations can be expressed in terms of these four primitive ones. Of course, in a 'production' implementation of the algorithms to be described below, we might well prefer an operation as important as $s\{y\} = x$ to have a direct rather than a contingent representation.

Extension of our schemata and algorithms to handle tuple operations is routine, and the necessary extensions are described in Section 4.

Intending to discover the places in a program P at which destructive pointer use will be legal, we begin by assuming that pointers are never used destructively. Thus, e.g., we assume that a copy of s is generated when the sets s+t or s-t are calculated. Each copying operation generates a new pointer, so that we reckon each operation such as s+t or s-t to be the *origin* of a new pointer; the pointer points to the data 'body' which represents s+t or s-t. On the other hand an operation such as {x} is assumed to make use of an old pointer and not to generate a copy of x. The set-theoretic sum s+t is assumed to generate a 'single-level' and not a 'full' copy of s. I.e., we generate a copy of the table representing s, but this table makes use of pointers to the existing members of s, which are themselves not copied. The consequences of a destructive calculation of s+t or s-t, which among its other effects propagates an old pointer rather than generating a new one, will be considered below.

In some cases, we may want to call explicitly for the formation of a copy of s. We assume that an operation *copy(s)* is available for this purpose. This is assumed to generate a 'single level', and not a 'full' copy in the sense just explained. For generating full copies explicitly, we might provide an operation *fullcopy(s)*. A 'fully independent copy' generating operation close to this is in fact called into play when a 'read x' operation brings in a SETL data object from an external medium. In what follows we shall represent operations like read x as schematic assignments x = *data*; where *data* is an operator that creates an object, perhaps composite, but having no pointer in common with any prior object.

2. Systems of Equations Describing the Transmission of Pointers.

Let i be an ivariable and let o be an ovariable of P. Then by *crthis(i)* (resp. *crthis(o)*) we mean the set of all variables which can create an object which at some moment in the execution of P becomes the current value of i (resp. o). If the value of i or o can be a set, then by *crmemb(i)* (resp. *crmemb(o)*) we mean the collection of all ivariables j whose values become incorporated as members into a set which at some moment in the execution of P becomes the current value of i (resp. o). We classify the operations occurring in P, and their associated variables o, as follows:

transfer operations: $o = i_1$;
null operations: $o = \underline{nl}$;
inclusion operations: $o = \{i_1\}$;
extraction operations: $o = \text{ᵹ}i_1$;
data operations: o = data;
setalgebraic operations: $o = i_1+i_2$; $o = i_1-i_2$ (for sets).
copy operations: $o = \text{copy}(i_1)$;
other algebraic operations: $o = i_1+i_2$; $o = i_1-i_2$; etc. (for

To define that one of these several classes to which the operation defining o belongs, we shall write an appropriate one of the predicates *transf(o), null(o), incl(o), extr(o), data(o), setalg(o), copyop(o), other(o).* When we wish to distinguish i_1+i_2 from i_1-i_2 in the *setalg* case, we shall write *setalgpls(o)* and *setalgmns(o)* respectively.

A system of equations defining the sets crthis(i), crthis(o), crmemb(i), and crmemb(o) results from the following considerations. Let i be an ivariable. Then

(1) crthis(i) = [+: o ∈ ud(i)] crthis(o);
crmemb(i) = [+: o ∈ ud(i)] crmemb(o);

Next, let o be an ovariable, and let i_1 (or i_1 and i_2) be the arguments of the operation defining o. Then o belongs to crthis(o). Moreover, if transf(o), then crthis(o) includes crthis(i_1). If extr(o), then crthis(o) includes [+: i∈crmemb(i_1)]crthis(i). Similarly, if transf(o), then crmemb(o) includes crmemb(i_1). If incl(o), then crmemb(o) is {i_1}. If algpls(o), then crmemb(i_1) + crmemb(i_2) is included in crmemb(o); if algmns(o), then crmemb(i_1) is included in crmemb(o). If copyop(o), then crmemb(o) includes crmemb(i_1). If data(o), then crmemb(o) is {nl}. Finally, if extr(o), then since the value of any i ∈ crmemb(i_1) may be transmitted to o, it follows that crmemb(o) = [+: i ∈ crmemb(i_1)]crmemb(i). The following equations in SETL notation express these facts.

```
(2) crthis(o) =      if transf(o) then crthis(arg1(o))
                        else if extr(o) then
                          [+: i∈crmemb(arg1(o))]crthis(i) else {o};
    crmemb(o) = if transf(o) or copyop(o) then crmemb(arg1(o))
                else if incl(o) then {arg1(o)}
                else if algpls(o) then crmemb(arg1(o))
                                     + crmemb(arg2(o))
                else if algmns(o) then crmemb(arg1(o))
                else if extr(o) then
                          [+: i ∈ crmemb(arg1(o))]crmemb(i)
                else if data(o) then {nl}
                else ω;
```

In these equations, *arg1* and *arg2* are understood to transform o into its first and second arguments. The system of equations (1) and (2) are, in an obvious sense, monotone in their right-hand sides, and can therefore be solved without difficulty by a convergent monotone iteration process.
We keep a 'workpile' which always contains those ivariables i and ovariables o for which some value crthis(i), crthis(o), crmemb(i), or crmemb(o) must be adjusted (necessarily upwards). Equations (1) and (2) are used when an adjustment must be made. The map du(o) is kept available, as is a map aux(o_1) sending each o_1 into the set of all ivariables i which are arguments of an operation of the form o = ∋i and for which $o_1 \in$ crmemb(i). Whenever crmemb(o_1) is adjusted, all $i \in$ du(o_1) are put back on the workpile for readjustment, as is every o which is the output of an operation with argument $i \in$ aux(o_1). Whenever crmemb(i) is adjusted, the output of the operation of which i is an argument is put back on the workpile. Moroever, if this operation has the form o = ∋i, then i is added to aux(o_1) for each o_1 newly added to crmemb(i). This adjustment process converges when the workpile has become null, at which point our determination of the maps *crmemb* and *crthis* will be complete.

By *crpart(i)* (resp. *crpart(o)*) we mean the set of all ovariables which can create an object which at some moment in the execution of P becomes a *part*, i.e. either an object identical with, or a member, or a member of a member, etc. of the current value of i (resp. o). Equations analogous to (1) and (2) can be stated for *crpart*, and these turn out to be almost the same as (1) and (2). The principal difference is simply that crpart(o) contains o unless transf(o) or extr(o), and that if incl(o), then crpart(o) includes both crthis(i_1) and crpart(i_1). This remark leads to the following equations:

```
(3) crpart(i) = [+: o ∈ ud(i)] crpart(o);
    crpart(o) = if transf(o) then crpart(argl(o))
                  else if extr(o) then
                    [+:il∈crmemb(argl(o))]crpart(il)
                  else {o} +
                  if incl(o) then crpart(argl(o))
                                  }
                  else if algpls(o) then crpart(argl(o))
                            + crpart(arg2(o));
                  else if algmns(o) or copyop(o)
                          then crpart(argl(o))
                else nl;
```

Once *crmemb* has been calculated, these equations can be solved by a straightforward monotone convergence procedure.

Let us now define *exsinthis(i)* (resp. *exsinthis(o)*) for ivariables i (resp. ovariables o) as follows: exsinthis(i) (resp. exsinthis(o)) is the set of all instructions executed since the creation of the pointer which is the current value of i (resp. o). We define *exsinmemb(i)* and *exsinpart(i)* (resp. *exsinpart(o)* and *exsinpart(o)*) in much the same way: exsinmemb(i) (resp. exsinmemb(o)) is the set of all instructions executed since the creation of a pointer which can be a member of the current value of i (resp. o); exsinpart(i) (resp. exsinpart(o)) is the set of all instructions executed since the creation of a pointer which can be a part of the current value of i (resp. o). The following considerations lead to equations analogous to (1-3) for these functions. Given an ivariable i representing an occurrence of v, let *exsinback(i)* denote the set of all operations which lie along some v-clear path beginning at the definition of v and terminating at i. Then

(4) exsinthis(i) = chainback(i) + [+: o ∈ ud(i)] exsinthis(o);
exsinmemb(i) = chainback(i) + [+: o ∈ ud(i)] exsinmemb(o);
exsinpart(i) = chainback(i) + [+: o ∈ ud(i)] exsinpart(o);

Next, let o be an ovariable, and let i_1 (or i_1 and i_2) be the arguments of the operation defining o. Then o belongs to exsinthis(o). Moreover, if transf(o), then exsinthis(o) includes exsinthis(i_1). If extr(o), then exsinthis(o) includes exsinmemb(i_1). Hence we have

(5) exsinthis(o) = {o} + if transf(o) then exsinthis (argl(o))
else if extr(o) then
exsinmemb(argl(o))else nl;

Similarly, o belongs to exsinmemb(o). In addition, if transf(o) or copyop(o), then exsinmemb(o) includes exsinmemb(i_1). If incl(o), then exsinmemb(o) includes exsinthis(i_1). If algpls(o), then exsinmemb(o) includes exsinmemb(i_1) + exsinmemb(i_2); if algmns(o), then exsinmemb(i_1) is included in exsinmemb(o). If extr(o), then the set exsinmemb(i) is included in exsinmemb(o); moreover, if i_1 is the variable of any operation in crmemb(o) (these i_1 necessarily belong to exsinmemb(i)) then the set exsinmemb(i_1) is included in exsinmemb(o).
Hence we have

(5') exsinmemb(o) = {o} +if transf(o) or copyop(o)
then exsinmemb(argl(o))

else if incl(o) then exsinthis(argl(o))

else if algpls(o) then exsinmemb(argl(o))+exsinmemb(arg2(o))
else if algmns(o) then exsinmemb(argl(o))
else if extr(o) then exsinmemb(argl(o))
+ [+: i_1 ∈ crmemb(argl(o))]exsinmemb(i_1)

else nl;

The quantity exsinpart(o) satisfies rather similar equations, which stand in the same relationship to (5') as (3) does to the second equation of (2). The main change to be noted is that, if incl(o), then exsinpart(o) includes both exsinthis(i_1) and exsinpart(i_1). Moreover, in the case extr(o), exsinpart(o) can be defined simply as {o} + exsinpart(i_1). Making these changes, and the other more routine changes that are necessary, we come to the following equations.

```
(5") exsinpart(o) = {o} + if transf(o) or copyop(o) or incl(o)
                          then exsinpart(arg1(o))

     else if algpls(o) then exsinpart(arg1(i))+exsinpart(arg2(o))
    else if algmns(o) then exsinpart(arg1(o))
     else if extr(o) then exsinpart(arg1(o))
     else nl;
```

Once *crmemb* has been calculated, the system (4,5,5',5") can be solved by a straightforward monotone convergence procedure.

We are now ready to determine the set $\ell(i)$ of all ovariables o whose current values might possibly incorporate the pointer p which is the current value of i. The operations which have been executed since p was created constitute the set P_i = exsinthis(i); the operations which may have created p constitute the set crthis(i). Given a subpart $\bar{P}$ of the program P, we define a 'relativised' *crpart* function $\text{crpart}_{\bar{P}}(o)$ as follows: o is an ovariable of $\bar{P}$, $\text{crpart}_{\bar{P}}(o)$ is the result of solving the equations (2,3,4) after replacing each occurrence of ud(i) by an occurrence of $\text{ud}(i) \cap \bar{P}$. Note that this has precisely the effect of ignoring all instructions that do not belong to $\bar{P}$. The condition that o should belong to $\ell(i)$ can then be written as

(6) $$\text{crpart}_{P_i}(o) \cap \text{crthis}(i) \neq \underline{n\ell} ,$$

which is of course equivalent to

(7) $$o \in \text{crpart}_{P_i}^{-1}[\text{crthis}(i)] .$$

Hence, we may state the following condition.

Destructive Use Condition. Let i be an ivariable of a SETL program P, and let P_i = exsinthis(i). Then if every o belonging to the set $\text{crpart}_{P_i}^{-1}[\text{crthis}(i)]$ is dead immediately before the i is used (where in describing an ovariable as dead we ignore its immediately following use as i) then i may be used destructively.

The following considerations illuminate an important aspect of the Destructive Use Condition. In the pages we have imagined that each pointer used at the SETL implementation level references some unique memory segment, and that a new pointer, with its own segment, is created each time a value creating operator (such as o = $\underline{n\ell}$; o = {i_1}; o = data; o = i_1+i_2;) rather than a value-extracting or value-transferring operator (i.e. o = i_1 or o = i_1) is executed. Now suppose that for some reason we decide to generate fewer pointers than would otherwise be required by re-using pointers where possible (in effect, performing an abstract kind of garbage collection). Plainly, a pointer p becomes available for re-use as soon as every object which could either be equal to p or contain p as a subpart is dead; we shall say in this situation that p is *totally dead*. Moreover, by re-using a pointer p which is available for re-use, instead of generating a new pointer, we do not change anything which could possibly be visible at the SETL level; such re-use is as fully 'transparent' as all other garbage-collector activity. Finally, we need not distinguish between re-used and newly generated pointers; for example,

an operator $o = \{i_1\}$ or $o = i_1+i_2$ may for all purposes be considered to generate the pointer p referencing the value $\{i_1\}$ or i_1+i_2 which is formed, and this irrespective of whether p is in fact re-used or newly generated. In particular by re-using totally dead pointers we cannot affect any of the information which enters into a decision as to whether the Destructive Use Condition is or is not satisfied at some particular point of a given SETL program P.

Suppose next that by applying the Destructive Use Condition we established that some operation of P, e.g. $o = i_1-i_2$, can and will be performed destructively. The Destructive Use Condition assures us that if p is any pointer which can appear as a value of i_1 , then, aside from its use as i_1, p is totally dead. Destructive use of p is essentially equivalent to re-use of p in the sense of the preceding paragraph; however, in re-using p we gain an efficiency advantage by exploiting the fact that the segment to which it points happens to contain a representation of the SETL value i_1. The argument of the preceding paragraph therefore shows that application of the Destructive Use Principle at one point in P does not affect its applicability at other points of P, i.e. that it may be applied *independently* to each of the instructions of P. This important addendum to the Destructive Use Principle will be used without explicit reference in what follows.

3. A New Examples.

Consider the code sequence

(1)
```
ℓ1:  s = nL;
ℓ2:  (while ... )
ℓ3:      s = s with x;
ℓ4:  end while;
ℓ5:  c = c with s;
```

Analyzing this example by the method described in the preceding section, we find that for the ivariable s appearing in line i3 (which for brevity we write as i:s:ℓ3) we have $\bar{P}$ = exsinthis(i:s:ℓ3) = {ℓ1,ℓ2,ℓ3}. Thus $crpart^{-1}_{\bar{P}}$[crthis(i:s:ℓ3)] = {o:s:ℓ1, o:s:ℓ3} (where by $o:v:\ell_j^{\bar{P}}$ we designate the appearance of v as an ovariable in line j) and then the Destructive Use Condition assumes us that i:s:ℓ3 can be used destructively.

On the other hand, in

(2)
```
ℓ1:  s = nℓ;
ℓ2:  (while ...)
ℓ3:     s = s with x;
ℓ4:     c = c with s;
ℓ5:  end while;
     ...
```

we have $\bar{P}$ = exsinthis(i:s:ℓ3) = {ℓ1,ℓ2,ℓ3,ℓ4}. Thus $crpart^{-1}_{\bar{P}}$[crthis(i:s:ℓ3)] = {o:s:ℓ1, o:s:ℓ3, o:c:ℓ4}, and since there is no reason why c should be dead before ℓ3 is executed, destructive use of i:s:ℓ3 is not possible.

Note that a SETL compiler incorporating the analytic procedures we have described could explicitly indicate the points at which it felt constrained to insert copying operations, and, by printing out the list of live members of $\bar{P}$, could indicate why thse operations were required. For example, the annotation 'copy s - because part of c in line ℓ4' could be attached automatically to line ℓ3 in (2).

Next consider the code

(3)
```
ℓ1:  s = t;
ℓ2:  (while ...)
ℓ3:     s = s with x;
ℓ4:  end while;
ℓ5:  c = c with s;
ℓ6:  d = d with t;
     ...
```

Analyzing this, we find that i:s:ℓ3 cannot be used destructively since this might change t which is used in ℓ6. Hence a copy operation is necessary. However, this copy operation can be moved out of the while loop, by changing (3) to the code sequence

(4)
```
ℓ1:   s = t;
ℓ1':  s = copy(s);
ℓ2:   (while ... )
ℓ3:      s = s with x;
ℓ4:   end while;
ℓ5:   c = c with s;
ℓ6:   d = d with t;
```

The analysis of (4) is very much like that of (1), and the possibility of using i:s:ℓ3 destructively follows by an argument like that given in connection with (1).

4. Tuple operations.

The analysis method described in section 2 ignores tuple operations. In the present section we shall remedy this shortcoming by describing the way in which the procedures of section 2 need to be extended if programs containing tuple operations are to be handled correctly. Of course, the algorithms of section 2 remain unchanged in general form; to handle tuple operations, we only need to change some of the details of these algorithms.

Our first task is to introduce a number of functions which are the analogs for tuples of the maps *exmemb* and *exsinmemb* of section 2. Let o be an ovariable of a SETL program P, and let n be an integer. Then *excomp(o,n)* is the set of all ivariables i' which can be transmitted to the n-th component of some (tuple) value of o (by not necessarily any component other than the n-th). Moreover, *exsomcomp(o)* is the set of all ivariables i' which can (as far as we know) be transmitted to any component of some (tuple) value of o. We define *exsincomp(o,n)* as the set of all instructions executed since the creation of a pointer which can be the n-th component of the current (tuple) value of o; and *exsinsomcomp(o)* as the set of all instructions executed since the creation of a pointer which can appear as some component of the current tuple value of o. Analogous functions *excomp(i,n)* *exsomcomp(i)*, *exsincomp(i,n)*, and *exsinsomcomp(i)* are defined for the ivariables i of P.

We continue to assume that P is available in schematised form. Now, however, we allow the following additional operations to appear in the schema representing P.

tuple-former operations:		$o = \langle i_1,\ldots,i_n\rangle$;
component extractors	:	$o = i_1(i_2)$ (i_1 a tuple)
subtuple extractors	:	$o = i_1(i_2{:}i_3)$ "
'tail' extractors	:	$o = i_1(i_2{:})$ "
component insertion	:	$o = [i_1(i_2) \longleftarrow i_3]$ (o a tuple)
tuple concatenation	:	$o = i_1 + i_2$ (i_1,i_2 tuples)

Note that the component insertion operation, which in order to conform to our general ivarable/ovariable conventions, we shall write as $o = [i_1(i_2) \longleftarrow i_3]$, is ordinarily written as v(n) = c; and ordinarily presumes destructive use of v (which is the i_1 of our schematic notation.) Moreover, we assume that before copy elimination is attempted for P, P has been subjected to a'typefinding' analysis of the kind described in A. Tenenbaum's thesis, so that the types of the variables appearing in P are known. It is of course not to be expected that our somewhat over-idealised assumptions will apply, in the precise form stated, to a full, production version of a copy optimisation program. Such a program would consequently be rather more complex than the still somewhat simplified algorithms which we are about to present. However, since in the present newsletter we wish to avoid writing out a long and highly detailed optimiser specification, we have little choice but to ignore these additional complications.

We shall write predicates *tform(o)*, *compex(o)*, *subtex(o)*, *tailex(o)*, *insa(o)*, and *concat(o)* to indicate that the operation defining o is of tuple-former, component extractor, subtuple extractor, 'tail' extractor, component insertion,or tuple concatenation type respectively.

The way in which we deal with certain tuple-related operations which have integer parameters will vary, depending on whether or not compile-time constant values or lengths of these parameters are known. To make this information available, we suppose a function *known(i)* to be available(as the result of a preliminary 'constant-propagation' process). The value known(i) is n if i is known at compile time to have the value n; otherwise known(i) is Ω. If the type of i is a tuple, known(i) gives its length if this is known, rather than i's value. The functions crcomp(i,n) and crcomp(o,n) will be recorded only when known(i) ne Ω (resp. known(o) ne Ω) and only when n ≤ known(i). In other cases, these functions are taken to have the nominal value nl.

The functions crcomp(i,n) and crsomcomp(i) satisfy the following equations:

(1) crcomp(i,n) = if known(i) is n ne Ω then
[+: o ∈ud(i)] crcomp(o,n)
else nl;
crsomcomp(i) = [+: o∈ud(i)] (crsomcomp(o) +
if known(i) ne Ω or known(o) eq Ω then nl
else [+: 1 ≤ m ≤ known(o)]crcomp(c,m));

To obtain equations for crcomp(o,n) and crsomcomp(o) we reason as follows. If tform(o), then crcomp(o,j) is $\{i_j\}$ for all j from 1 to the number of input parameters of o. Moreover, crthis(o) = {o}. If concat(o), then crthis(o) = {o}. Moreover, if known(i_1) = n, we have crcomp(o,j) = crcomp(i_1,j) for 1 ≤ j ≤ n, crcomp(o,j) = crcomp(i_2,j-n) for 1 ≤ j ≤ known(i_2). However, if known(i_1) = Ω, then crcomp(i_1,j) will always be nl, and crsomcomp(o) is the union of crsomcomp(i_1), crsomcomp(i_2), and of all the non-null sets among crcomp(i_1,j) and crcomp(i_2,j). For the 'tail' extraction operator, and assuming that known(i_2) = n and known(i_1) ≠ Ω, we have

crcomp(o,j) = crcomp(i_1,j - n) for j > n; crsomcomp(o) = crsomcomp(i_1). If known(i_2) is Ω but known(i_1) ≠ Ω,then crsomcomp(o) = crsomcomp(i_1) + [+: 1 ≤ j ≤ known(i_1)] crcomp(i_1,j). If both known (i_1) and known (i_2) are Ω, then we have simply crsomcomp(o) = crsomcomp(i_1). Similar relations hold if subtext(o); relevant details appear in the formal equations for crsomcomp(o) and crcomp(o,n) written below. If inxa(o), known(i_2) is n ne Ω, and known(i_1) ne Ω, then we have crcomp(o,n) = {i_3} while crcomp(o,n) = crcomp(i_1,m) for m ≠ j and crsomcomp(o) = crsomcomp(i_1). If known(i_1) ne Ω but known(i_2)eq Ω, then crcomp(o,m) = crcomp(i_1,m) for all m, while crsomcomp(o) = crsomcomp(i_1) + {i_3}. Details concerning the cases in which known(i_1) eq Ω appear in the formal equations written below.

The component extractor case,i.e. the case compex(o), affords complications much like those already encountered in the extr(o) case of section 2. Considering the compex(o) case, suppose first that known(i_2) is n ne Ω and that known(il) ne Ω.

Then since crcomp(o,n) is the set of all ivariables which can be transmitted to the n-th component of the value of o, while crsomcomp(n) is the set of all ivariables which can be transmitted to some (variable-or-unknown) component of the value of o, we have

(2) crthis(o) = [+: i ∈ crcomp(i_1,n) + crsomcomp(i_1)] crthis(i).

For much the same reason, we have

(2') crmemb(o) = [+: i ∈ crcomp(i_1,n) + crsomcomp(i_1)] crmemb(i);

If known(o) is m ne Ω, we have

(2'') crcomp(o,j) = [+: i ∈ crcomp(i_1,n) + crsomcomp(i_1)] crcomp(i,j)

and

(2") crsomcomp(o) = [+: i ∈ crcomp(i_1,n) + crsomcomp(i_1)]crsomcomp(i),

both for $1 \leq j \leq m$. On the other hand, if known (o) eq Ω, then all crcomp(o,j) are taken to be Ω, and we have

(3) crsomcomp(o) = [+: i ∈ crcomp(i_1,n) + crsomcomp(i_1)]
(crsomcomp(i) +
if known(i) is m eq Ω then nl else
[+: 1 ≤ j ≤ m] crcomp(i,j)) .

Corresponding details for the compex(o) cases for which either known(i_1) eq Ω or known(i_1) eq Ω are found in the formal equations written below.

This survey should suffice as introduction to the following equations for the functions crcomp(o) and crsomcomp(o), and to the following revised equations for the functions *crthis*, *crmemb*, and *crpart*, all of which we now proceed to give.

First we give the equation for *crthis*.

```
(4)  crthis(o) =if null(o) or incl(o) or data(o) or setalg(o)
                  or copyop(o) or other(o) or tform(o) or
                  or subtex(o) or tailex(o) or inxa(o) or concat(o)
                then {o}
                else if transf(o) then crthis(argl(o))
                else if extr(o) then
                  [+: i ∈ crmemb(argl(o))]crthis(i)
                else /* if compex(o) then */
                if known(argl(o) is il) is tuplen ne Ω then
                  if known(arg2(o)) is inx ne Ω then
                    [+: i ∈ crcomp(il,inx)+crsomcomp(il)]crthis(i)
                  else /* if inx eq Ω then */
                    [+: i ∈ [+: 1 ≤ n ≤ tuplen]crcomp(il,n)
                          + crsomcomp(il)]crthis(i)
                  else /* if tuplen eq Ω then */
                          [+: i∈crsomcomp(il)]crthis(i);
```

The equation for *crmemb* is as follows:

```
(5) crmemb(o) = if null(o) or other(o) or tform(o) or subtex(o)
                  or tailex(o) or inxa(o) or concat(o)
                then nl
                else if incl(o) then crthis(arg1(o))
                else if algpls(o) then crmemb(arg1(o))
                                     + crmemb(arg2(o))
                else if algmns(o) or transf(o) or copyop(o)
                                     then crmemb(arg1(o))
                else if data(o) then {nl}
                else if extr(o) then
                  [+: i ∈ crmemb(arg1(o))] crmemb(i)
                else /* if compex(o) then */
                if known(arg1(o)is il) is tuplen ne Ω then
                  if known(arg2(o)) is inx ne Ω then
                    [+: i ∈ crcomp(il,inx) + crsomcomp(il)]crmemb(i)
                  else /* if inx eq Ω then */
                    [+: i ∈ [+: 1 ≤ n ≤ tuplen]crcomp(il,n)
                                     + crsomcomp(il)]crmemb(i)
                else /* if tuplen eq Ω */ then
                                [+: i∈crsomcomp(il)]crmemb(i);
```

Next we give the equation for *crcomp(o,j)*. This equation is used only if the type of o is a tuple and known(o) ne Ω, in which case we have one equation for each 1 ≤ j ≤ known(o). Note that the function arg(o,j) which appears below is assumed to return the j-th argument of the operation defining o; this argument is of course an ivariable.

```
(6)  crcomp(o,j) = if tform(o) then crthis(arg(o,j))
                   else if subtex(o) or tailex(o) then
                     crcomp(i1,j+known(i2))
                   else if inxa(o) then
                     if known(arg2(o)) is inx ne Ω then
                       if j eq inx then crthis(arg3(o))
                         else crcomp(arg1(o),j)
                      else /* if inx eq Ω then */ nl
                   else if concat(o) then
                     if j le known(arg1(o)) is len1 then
                                        then crcomp(arg1(o),j)
                     else crcomp(arg2(o)-len1)
                   else if transf(o) or copyop(o)
                                        then crcomp(arg1(o),j)
                   else if data(o) then {nl}
                   else if extr(o) then
                     [+: i ∈ crmemb(arg1(o))]crcomp(i,j)
                   else /* if compex(o) then */
                   if known(arg1(o) is i1) is tuplen ne Ω then
                     if known(arg2(o)) is inx ne Ω then
                       [+: i ∈ crcomp(i1,inx) + crsomcomp(i1) ]
                                              crcomp(i,j)
                     else /* if inx eq Ω then */
                       [+: i ∈ [+: 1 ≤ n ≤ tuplen]crcomp(i1,n)
                            + crsomcomp(i1)]crcomp(i,j)
                   else /* if tuplen eq Ω then */
                     [+: i ∈ crsomcomp(i1)] crcomp(i,j);
```

Now we give the equation for *crsomcomp(o)*. This equation is used only if the value of o is a tuple, and is complicated by the need to deal with various subcases which arise depending on whether or not the lengths of o and of the several ivariables defining o are known.

```
(7)  crsomcomp(o) =
      if tform(o) then nℓ
        else if data(o) then {nℓ}
        else if subtex(o) then
          if known(arg1(o) is i1) is tuplen eq Ω then
                  crsomcomp(i1)
          else /* if tuplen ne Ω then */
            if known(arg2(o)) is inx1 eq Ω then
              crsomcomp(i1) + [+: 1 ≤ n ≤ tuplen]crcomp(i1,n)
            else /* if inx1 ne Ω then */
              if known(arg3(o)) eq Ω then
              crsomcomp(i1) + [+: inx1 ≤ n ≤ tuplen]crcomp(i1,n)
              else crsomcomp(i1)
        else if tailex(o) then
          if known(arg1(o) is i1) is tuplen eq Ω then
                  crsomcomp(i1)
          else /* if tuplen ne Ω then */
            if known(arg2(o)) is inx1 eq Ω then
            crsomcomp(i1) + [+: 1 ≤ n ≤ tuplen] crcomp(i1,n)
            else /* if inx1 ne Ω then */ crsomcomp(i1)

        else if inxa(o) then
          if known(arg1(o) is i1) eq Ω or known(arg2(o)) eq Ω then
                  crsomcomp(i1) +  crthis(arg3(o))
          else crsomcomp(i1)
        else if concat(o) then crsomcomp(arg1(o) is i1)
                               + crsomcomp(arg2(o) is i2) +
          if known(i1) is len1 eq Ω then
            if known(i2) is len2 eq Ω then nℓ
            else [+: 1 ≤ n ≤ len2] crcomp(i2,n)
          else /* if len1 ne Ω then */
            if known(i2) ne Ω then nℓ
            else [+: 1 ≤ n ≤ len1] crcomp(i1,n)
```

```
else if transf(o) or copyop(o) then crsomcomp(arg1(o))
else if extr(o) then
  [+: i ∈ crmemb(arg1(o))](crsomcomp(i) +
     if known(o) eq Ω and known(i) is ilen ne Ω then
       [+: 1 ≤ n ≤ ilen]crcomp(i,n) else nl)
else /* if compex(o) then */
  if known(arg1(o) is il) is tuplen ne Ω then
    if known(arg2(o)) is inx ne Ω then
      [+: i ∈ crcomp(il,inx)+crsomcomp(il)](crsomcomp(i)+
        if known(o) eq Ω and known(i) is ilen ne Ω then
          [+: 1 ≤ n ≤ ilen]crcomp(i,n) else nl)
    else /* if inx eq Ω then */
      [+: i ∈ [+: 1 ≤ m ≤ tuplen]crcomp(il,m)
                    + crsomcomp(il)](crsomcomp(i) +
        if known(o) eq Ω and known(i) is ilen ne Ω then
          [+: 1 ≤ n ≤ ilen]crcomp(i,n) else nl)
else /* if tuplen eq Ω then */
  [+: i ∈ crsomcomp(il)](crsomcomp(i)+
    if known(o) eq Ω and known(i) is ilen ne Ω then
      [+: 1 ≤ n ≤ ilen] crcomp(i,n) else nl);
```

To allow for the existence of tuple operations, we must revise the definition of the sets *crpart* as follows (cf. the paragraph preceding formula (3) of Section 2): By *crpart(i)* (resp. *crpart(o)*) we mean the set of all ovariables which can create a pointer which at some moment in the execution of P becomes a *part*, i.e. either a pointer identical with, or a member pointer, or a component pointer, or a member of a member, member of a component, component of a member etc. pointer of the current value of i (resp. o). Revised equations for *crpart* are as follows:

```
(8)  crpart(i) = [+: o ∈ ud(i)]crpart(o);
     crpart(o) = if transf(o) then crpart(arg1(o))
                 else if extr(o) then
                   [+: i ∈ crmemb(arg1(o))]crpart(i)
                 else if compex(o) then
                   if known(arg1(o) is il) is tuplen ne Ω then
                     if known(arg(2)) is inx ne Ω then
                     [+: i ∈ crcomp(il,inx)+
                              +crsomcomp(il)]crpart(i)
                     else /* if inx eq Ω */ then
                  [+: i∈[+: 1≤n≤tuplen]crcomp(il,n)
                              + crsomcomp(il)]crpart(i)
                   else /* if tuplen eq Ω */ then
                   [+: i ∈ crsomcomp(il)]crpart(i)
                  else {o} +
                  if incl(o) then crpart(arg1(o))
                  else if algpls(o) or concat(o)
                    crpart(arg1(o)) + crpart(arg2(o))
                  else if algmns(o) or copyop(o) then
                              crpart(arg1(o))
                  else if subtex(o) then
                    if known(arg1(o) is il) is tuplen eq Ω
                      or known(arg2(o)) is inx eq Ω then crpart(il)
                    else /* if tuplen ne Ω and inx  ne Ω then */
                      if known(arg3(o)) is inxhi eq Ω then
                [+: i∈crsomcomp(il)+[+:inx≤n≤tuplen]crcomp(il,n)]
                                                      crpart(i)
                    else /* if inxhi ne Ω then */
                [+: i∈crsomcomp(il)+[+: inx ≤ n ≤ inxhi]crcomp(il,n)]
                                                      crpart(i)
                   else if tailex(o) then
                     if known(arg1(o) is il) is tuplen eq Ω
                       or known(arg2(o)) is inx eq Ω then crpart(il)
```

```
                else /* if tuplen ne Ω and int ne Ω then */
            [+: i∈crsomcomp(il)+[+: inx≤n≤tuplen]crcomp(il,n)]
                                                crpart(i)
          else if inxa(o) then
            if known(arg1(o) is il) is tuplen eq Ω
              or known(arg2(o)) is inx eq Ω then
                crpart(il) + crpart(arg3(o))
            else
[+: i∈crsomcomp(il)+[+: 1≤n≤tuplen|n ne inx]crcomp(il,n)]crpart(i)
                          + crpart(arg3(o))
     else if data(o) then {nl}
     else if tform(o) then
          [+: 1≤n≤nargs(o)]crpart(arg(o,j))
          /* where nargs(o) is the number of arguments
             of the tuple-forming operation defining o */
     else /* for other operators */ nl;
```

The equations for *crthis*, *crmemb*, *crpart*, *crcomp*, and *crsomcomp* stated in the present section can be solved by a procedure hardly differing from the iterative process outlined in Section 2. The Destructive Use Condition retains its validity in the presence of tuple operations.

The function *exsinthis(i)* appears in the statement of the Destructive Use Condition and for this reason we continue the present section by giving the equations required to calculate *exsinthis*. In this function there appear several other functions:

exsinmemb, which is the set of all instructions executed since the creation of a pointer which is a member of the current value of an ivariable (or ovariable); *exsinsomcomp*, which is the set of all instructions executed since the creation of a pointer which can appear as some (unknown) component of the current value of (an i- or o-) variable; and *exsincomp(o,j)* (resp. *exsincomp(i,j)*) which is the set of all instructions executed since the creation of a pointer which can appear as the j-th component of the current value of o (resp. i). These various quantities obey equations rather like those for *orthis*, *ormemb*, *orsomcomp*, etc. which have just been written. We shall for this reason not give a full set of equations; instead, we shall confine ourselves to writing out the equations for *exsinthis*, *exsinmemb*, and *exsincomp(o,j)*. Note that *exsincomp(o,j)* and *exsincomp(i,j)* are defined only when known(o) ne Ω (resp. known(i) ne Ω); in other cases, *exsincomp(o,j)* and *exsincomp(i,j)* have the nominal value nl. For an ivariable i we have

(9) exsinthis(i) = chainback(i) + [+: o∈ud(i)]exsinthis(o);
exsinmemb(i) = chainback(i) + [+: o∈ud(i)]exsinmemb(o);
exsincomp(i,j)=chainback(i) + [+: o∈ud(i)]exsincomp(o,j);

(Cf. equation (4) of Section 2, and also the remarks preceding that equation.)

Next, let o be an ovariable, and let i_1 (or i_1 and i_2) be the arguments of the operation defining o. Then o belongs to exsinthis(o). If transf(o), then exsinthis(o) includes exsinthis(i_1). If extx(o), then exsinthis(o) includes exsinmemb(i_1). If compex(o) and known(i_1) ne Ω, while known(i_2) is n ne Ω, then exsinthis(o) includes exsincomp(i_1,n). Other cases will arise depending in the values of known(i_1) and known(i_2); details of these cases appear in the equations which follow. Other operations, such as indexed assignment, sub-tuple extraction, and concatenation create new SETL objects and hence make no special contribution to exsinthis(o). All in all we have

```
(10)  exsinthis(o) = {o} + if transf(o) then exsinthis(argl(o))
                           else if extr(o) then
                                exsinmemb(argl(o))
                           else if compex(o) then
                      if known(argl(o) is tupl) is tuplen eq Ω then
                         exsinsomcomp(tupl)
                      else if known(arg2(o)) is inx eq Ω then
                         exsincomp(tupl,inx)
                      else exsinsomcomp(tupl)+
                           [+: 1≤n≤tuplen] exsincomp(tupl,n)
                         else /* for other operations */ nl;
```

Next we give an equation for *exsinmemb(o)*. The equation which we give is justified by reasoning like that which precedes equation (5') of section 2; we leave it to the reader to work out necessary details:

```
(10')  exsinmemb(o) = {o} + if transf(o) or copyop(o) or algmns(o)
                              then exsinmemb(argl(o))
   else if incl(o) then exsinthis(argl(o))
   else if algpls(o) then exsinmemb(argl(o))+exsinmemb(arg2(o))
   else if extr(o) then exsinmemb(argl(o))
        + [+: i ∈ crmemb(argl(o))] exsinmemb(i)
   else if compex(o) then
        if known(argl(o) is tupl) is tuplen eq Ω then
             exsinsomcomp(tupl)+[+: i∈crsomcomp(tupl)]exsinmemb(i)
        else if known(arg2(o)) is inx eq Ω then
             exsincomp(tupl,inx)+
                  [+: i∈crcomp(tupl,inx)]exsinmemb(i)
        else exsinsomcomp(tupl)+
                  [+: i∈crsomcomp(tupl)] exsinmemb(i)
   [+: 1≤n≤tuplen](exsincomp(tupl,n)+
                  [+: i∈crcomp(tupl,n)]  exsinmemb(i))
   else /* for other operators */ nl;
```

Finally, we give an equationfor exsincomp(o,n). As already stated, this function is only defined when known(o) ne Ω and only for n ≤ known(o). The equation which we give can be justified by reasoning adapted from that which precedes equation (5') of Section 2; however we leave all details of this reasoning to the reader.

```
(11)  exsincomp(o,n) = {o} + if transf(o) or copyop(o)
                             then exsincomp(argl(o),n)
   else if tform(o) then exsinthis(arg(o,n))
   else if tailex(o) or subtex(o) then
                             exsincomp(argl(o), n + known(arg2(o)))
   else if concat(o) then
        if n le (known(argl(o) is i) is ilen then exsincomp(i,n)
        else exsincomp(arg2(o), n - ilen)
   else if inxa(o) then
        if known(arg2(o)) is inx eq Ω then nΩ
        else if n ne inx then exsincomp(argl(o),n)
             else exsinthis(arg3(o))
   else if extr(o) then exsinmemb(argl(o))
        + [+: i ∈ ormemb(argl(o))]exsincomp(i,n)
   else if compex(o) then
        if known(argl(o) is tupl) is tuplen eq Ω then
             exsinsomcomp(tupl)+[+: i∈crsomcomp(tupl)]exsincomp(i,n)
        else if known(arg2(o)) is inx ne Ω then
             exsincomp(tupl,inx)+[+: i∈crcomp(tupl,inx)]
                                                  exsincomp(i,n)
        else exsinsomcomp(tupl)+[+: i∈crsomcomp(tupl)]exsincomp(i,n)
   - [+: 1≤j≤tuplen(exsinsomcomp(tupl))+[+: i∈crcomp(tupl,j)]
                                                  exsincomp(i,n)]
   else /* for other operators */ nΩ;
```

5. Additional examples.

Next, in order to get some idea of the advantage which our copy optimizations might secure in typical cases, we examine a number of codes taken from O.P. II. The first is the Cocke-Earley 'nodal span' parse, in the form given on page 161-162. Here the algorithm is

```
define nodparse(input,gram,root,syntypes,spans,divlis,amb);
todo = nl; divlis = nl; spans = {<2,s,1>, s∈syntypes{input(1)}};
  (1 ≤ ∀n ≤ #input)
    todo = {<n+1,s,n>, s ∈ syntypes{input(n)}};
    spans = spans + todo;
      (while todo ne nl)
      next from todo;
      <end,typ2,mid> = next;
        (∀spend∈spans{mid}, type∈gram{hd spend is typ1,typ2})
          newsp = <end,type,spend(2)>;
          <newsp,mid,typ1,typ2> in divlis;
          if n newsp ∈ spans then
            newsp in spans;  newsp in todo;
          end if;
        end ∀spend;
      end while;
  end ∀n;
  /* check on grammaticality */
  if n (<#input+1,root,1> is topspan) ∈ spans then
    <spans,divlis,amb> = <nl,nl,f>;  return;
  end if;
  /* else clean up set of spans and determine ambituity */
  spans = nl;  amb = f;  getdescs(topspan);
  /* clean division list */
  divlis = {d ∈ divlis | hd d ∈ spans};
  return;
  end nodparse;
```

Destructive use of a variable is potentially possible in the operation spans = spans + todo; next from todo; <newsp,mid,typ1,typ2> in divlis; newsp in spans; newsp in todo; . The Destructive Use Condition of Section 2 can be seen to validate each of these destructive uses; all that is involved is the rather trivial observation that neither *todo*, *divlis*, or *spans* ever becomes an element of any larger composite object.

Next we examine the 'graph ordering' algorithm given on p. 266 of O.P. II. The code for this is

```
definef graphord(nodes, entry);
/* the successor map cesor is assumed to be global */
order = <entry>;
mark = {<entry,t>};
jlast = 1 /* jlast is highest numbered node from which
                                   new path may begin */;
(while jlast ≥ ∃j ≥ 1, last ∈ cesor(order(j))|mark(last) ne t)
/* start new path */
   path = <last>;  mark(last) = t;
/* and extend as far as possible */
   (while ∃ next ∈ cesor(last)|mark(next) ne t)
      path = path + <next>;
      mark(next) = t;  last = next;
   end while ∃;
/* insert path after jth node in order */
   order = order(1:j) + path + order(j+1:);
   jlast = j + #path + 1;
/* note path(#path) has no unmarked successors */
end while jlast;
return order;
end graphord;
```

Here the potentially destructive operations are path = path+<next>; and mark(next) = t; and both of these are admissible by the Destructive Use Condition, simply because neither *path* nor *mark* ever becomes an element of a larger composite.

Next we consider a fragment from the set of 'interval-finding' routines given in O.P. II, pp. 269 ff. The code in question is as follows:

```
definef interval(nodes,x);
/* npreds, followers  and cesor are assumed to be global */
/* count the number of predecessors 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;
```

We ignore any special problems having to do with 'cross subroutine' optimization, thus in effect assuming that these two functions are in some appropriate way consolidated into one.

The operations containing potentially destructive variable uses are: npreds(y) = npreds(y)+1; count(x) = npreds(x); z out followers; count(y) = count(y) + 1; y in followers; (and now passing to the subroutine *intervals*) node from seen; follow(i) = followers; seen = seen+followers;. With the exception of z out followers and y in followers, the legitimacy of all these destructive uses is obvious from the fact that neither *npreds*, *count*, *seen*, nor *follow* are made elements of any larger objects. Destructive use of followers in z out followers and y in followers is also justified, but somewhat more of the force of the Destructive Use Principle is needed to establish this. Specifically, we must note that from the initial definition of *followers* up to the (dynamic) occurrence of z out followers no instruction making *followers* part of a more compound object is executed. A similar remark applies to the occurrence of y in followers.

Note that the set {y∈followers|npreds(y) eq count(y)} is a good candidate for elimination by Earley's method of 'iterator inversion', i.e. by set-theoretic strength reduction.

6. A General Remark Concerning the Mappings Described in the Preceeding Pages

The mapping *crthis* discussed in the preceeding pages is analogous to the mapping *ud* which appears in conventional data-flow analysis, i.e., to the map which chains each use of a variable to those definitions which can set the value of the variable. However, whereas in calculating the mapping *ud* we only link an ivariable i to an ovariable o if there is a path leading from o to i which is free of redefinitions of the variable common to o and i, the mapping *crthis* will link i to o wherever the object x created at o can be transmitted to i, either along such a path or indirectly through any chain of operations which embed x into some nested collection of sets and tuples and then later extract it. Thus *crthis* expresses a considerably more extensive concept of value transmission than does *ud*.

The inverse map $crthis^{-1}$ extends the definition-to-use chaining map *du* ordinary data flow analysis in a similar way. In chosing data representations for the objects appearing in SETL programs we will often need to know all the operations applied to the object created at a given ovariable o. Once the map $crthis^{-1}$ has been calculated, this information is easily obtained: we have only to find all the ivariables in $crthis^{-1}(o)$, and note the operations to which these ivariables are arguments.

It will be seen in a later newsletter that by regarding the ovariables of a program as nodes in a graph G whose edges are defined by the mappings *crmemb*, *crcomp*, *crsomcomp* discussed above and then by analysing this graph we can hope to gain some idea of the logical data-types in terms of which a program is organised. In particular, recursive data types correspond to loops in G.

7. Paths Along Which Variables and SETL Values are Live.

For certain purposes one will need to know not only the pattern in which ivariables are chained to ovariables as uses and vice-versa, but even the specific paths along which an ovariable o is connected to an ivariable which uses the value generated by o; these are the so-called *live paths* for o. In optimising languages of the FORTRAN level, information of this type can be used in generating register allocations by a packing process; applications to SETL of information of this same type will be described in section 8 and 9 below. In a conventional data flow analysis, a live path calculation will have two functions, which we shall call *pud* and *pdu*, as result. The function *pdu* maps each ovariable o into the set of all program edges which belong to a path connecting o to one of its uses; the set *pud* maps each ivariable i into the set of all edges belonging to a path connecting i to one of its defining ovariables. These two functions can easily be expressed in terms of two functions which are in any case apt to be calculated during conventional data flow analysis. The first of these functions, *reaches(b)*, gives the set of all ovariables o whose values would be used in the basic block b if b contained a use of the variable v of o; the second of these functions, *live(b)*, gives the set of ivariables which are live on entrance to the basic block b. Then the edge e starting at a block b_1 and ending at b_2 belongs to pdu(o) if some ivariable i involving the same variable v as o satisfies i ∈ live(b_2) and if either o ∈reaches(b_1) and b_1 is free of redefinitions of o, or if o is the last target occurence of v in b_1. Similarly, e belongs to pud(i) if i ∈ live(b_1) and if there exists an ovariable o with variable b such that either o ∈ reaches(b_1) and b_1 is free of redefinitions of o, or if o is the last target occurence of v in b_1.

In the present section we will show how various mappings related to *pud* and *pdu* can be calculated. These mappings generalise *pud* and *pdu* in much the same way that the maps *erthis*, *erpart*, *erhold*, etc, generalise *ud* (compare the remarks made in the preceeding section.) The functions in which we shall be interested are as follows:

a. *perthis(i,o)* defines the set of all program graph edges which can belong to an execution-time path connecting an ivariable i with an ovariable o which creates a SETL object which reappears as the value of i.

b. *permemb(i,o)* defines the set of all program graph edges which can belong to an execution-time path connecting an ivariable i with an ovariable o which creates a SETL object which is a member of a set appearing as the value of i.

c. *perpart(i,o)* defines the set of all program graph edges which can belong to an execution-time path connecting an ivariable i with an ovariable o which creates a SETL object which is a part of (i.e. a member of, a member of, a member of, etc.) a set appearing as the value of v.

Note that in the present section, as in section 2 above, we simplify our discussion by ignoring tuple operations completely, and by assuming that the only four set-theoretic operations which appear in our schematised programs are s+t, s-t, {x}, and ∋s. Equations for *perthis*, *permemb*, *perpart* etc. could of course be developed for full SETL including tuple operations: the general structure of these equations would be similar to the simplified equations which will be written below. However, not wishing to write out yet another lengthy group of set-theoretic formulae, we shall not present these fuller equations.

Note however that in treating tuple operations we would introduce functions *persomcomp* and *percomp* in addition to the functions *perthis*, *permemb*, and *perpart* appearing below.

In writing equations (1-3) below we use various major and auxiliary functions introduced in section 2, including the functions *arg1* and *arg2* which transform an ovariable o into its first and second arguments.

The equations which are now to be given are justified by the following reasoning. First consider *perthis*, and a program path p which leads from an ovariable o to an ivariable i at which the value x created by o reappears. If x is re-created by extraction somewhere along the path, there will be some last point α at which an extraction operator is applied to obtain x ; the operand of this extraction operator must of course be some set of which x is a member. Past α, the value x can only be transmitted, by transmission operations v = expn. If such operations appear at the end of p, then p can be decomposed into two shorter paths, the second of which is a path from a transmission operator to i, and the first of which is a path from o to an ivariable i_1 at which x reappears. If no such operations occur at the end of p, then α is the last relevant operation in p, and p decomposes into the part of p preceeding α, plus a part following α which is free of operations relevant to x. This leads us to the following equation for *perthis*:

(1) $[+: o_1 \in ud(i) | transf(o)](pud(i,o_1) + perthis(arg1(o_1),o))$
$+[+: o_1 \in ud(i) | extr(o)](pud(i,o_1) + permemb(arg1(o_1),o))$.

Next, consider *permemb*, and a program path p which leads from an ovarible o to an ivariable i at which a set s containing the value x created by o appears.

Let α denote the position of the operation along p which creates the set s. Let o_1 be the ovariable of this operation. Then the part of p which follows α consists entirely of edges belonging to pcrthis(i,o_1). The operation whose value defines o_1 must be either a copy operation, a setalgebraic operation, or an inclusion operation. In the first two of these three possible cases, x must be a member of an appropriate operation argument; if o_1 is the target ovariable of an inclusion operator with argument ivariable i_1, then x must be a possible value of i_1. This justifies the following equation for *pcrmemb*:

(2) [+: $o_1 \in$ crthis(i)|copy(o_1) or setalg(o_1)](pcrthis(i,o_1) + pcrmemb(arg1(o_1),o))
+[+: $o_1 \in$crthis(i)|algpls(o_1)](pcrthis(i,o_1) + pcrmemb(arg2(o_1),o)
+[+: $o_1 \in$crthis(i)|incl(o_1)](pcrthis(i,o_1) + pcrthis(arg1(o_1),o))

The function *perpart* satisfies a similar equation which can be justified by a similar argument. Consider a program path p which leads from an ovariable o to an ivariable i at which there appears a set s containing as a part the value x created by o. Let α denote the position of the operation along p which creates the set s, and let o_1 be the ovariable of this operation. Then the part of p which follows α consists entirely of edges belonging to pcrthis(i,o_1). The operation whose value defines o_1 must be either a copy operation, a setalgebraic operation, or an inclusion operation. In the first two of these three possible cases, x must be a part of an appropriate operation argument; if o_1 is the target ovariable of an inclusion operator x must be either a possible member of a possible value of i_1 or a possible part of such a value. Hence we deduce the following equation for *perpart*:

(3) [+: $o_1 \in$crthis(i)|copy(o_1) or setalg(o_1)](pcrthis(i,o_1) + perpart(arg1(o_1),o))
+[+: $o_1 \in$crthis(i)|algpls(o_1)](pcrthis(i,o_1) + perpart(arg2(o_1),o)

$$+[+:\ o_1\ \varepsilon crthis(i)|incl(o_1)](pcrthis(i,o_1) + pcrpart(argl(o_1),o) + pcrthis(argl(o_1),o)).$$

The system (1-2-3) of equations, as well as the more elaborate system which will replace it when tuples and tuple operations are allowed to enter our considerations, can be solved routinely by the method of monotone convergence described in an earlier section. The functions *perthis*, *permemb*, and *perpart* obtained by solving this system express useful relationships between dataflow and control flow in SETL programs. The next two sections of the present newsletter exhibit some of the uses which these functions have.

8. Replacing Blank Atoms by Pointers.

To mimic pointer semantics in SETL, one can use blank atoms, introduce a global mapping *pointsto*, and regard each pointer as a blank atom which can only be used to index this mapping. Of course, SETL permits more general pointer constructions, e.g. the SETL user can introduce many such 'pointing' functions into a single program. There is however one aspect of the way in which pointers are ordinarily used in pointer-oriented languages which is missed in an unoptimised variant of SETL. In a pointer oriented language, only objects which can be reached through a chain of relationships p_j = pointsto(p_{j-1}) starting from the value p_1 of some explicit pointer variable (or perhaps from a recursively stacked incarnation of such a variable) are live; all other objects are recognisably dead and can be garbage-collected. In unoptimised SETL this fact will be missed, since the SETL compiler will assume that expressions such as *# pointsto*, *pointsto* eq *s*, etc, might have to be evaluated; and of course garbage collection as applied in pointer languages can change the values of these expressions.

Suitable optimisation can relieve this difficulty and convert SETL programs which mimic pointer constructions into codes which actually use pointers. The following approach can be used: we introduce an implementation-level object type *pointer*, flagged in some recognisable way. In addition to its tag field, a pointer will contain a machine address field referencing a block in the heap; two pointers will only be equal if their tags and addresses are equal. Then, whenever a blank atom x is created by a call to newat, we must decide whether to create it as a pointer (for which a block of storage will at the same time be allocated) or to create it in the standard SETL form. For the former choice to be made, we demand the following:

a) Among the operations in which the value x appears are certain indexed retrievals $f_j(x)$ and indexed assignments $f_j(x) = y$, for which the following conditions are satisfied:

b) The values of the variables $f_j, j = 1,\ldots,m$ appearing as first argument in these indexed retrievals and assignments are never themselves made components of any vector or members of any set.

c) The only operations applied to the mappings f_j are retrievals $f_j(y)$, indexed assignments $f_j(y) = z$, and assignment operations $f_j = w$.

d) Let o be the target ovariable of the call to newat which creates the blank atom x. Let i be an ivariable appearing in the context $f_j(i)$. Then we insist that no assignment $f_j = w$ having f_j as target variable can belong to the set perthis(o,i).

Note that condition (d) implies that the value $f_j(x)$ will never be calculated after an assignment $f_j = z$ has been made. This allows us to leave an 'obsolete' value $f_j(x)$ recorded in the block to which x points.

If the target ovariable o of a newat call satisfies conditions (a-d), then the atom x generated by the call can be created as a pointer, and a heap block hb sufficient to record the value of each function f_j satisfying conditions (a-d) (in regard to x) can be allocated; of course, x should point to hb. If this is done, values $f_j(x)$ can be retrieved by extracting fields from hb, essentially as if $f_j(x)$ were transformed into pointsto(x)(n_j). Note that this mode of access need not be used for every element of the domain D of f_j, but only for those which are pointers created by a particular newat call. For other elements of D, the standard SETL hashing scheme can be used to access $f_j(x)$. Of course, maximum advantage is obtained when all elements of the domain of a function f_j are pointers of the same kind, since in this case a minimum number of conditional transfers appear in the code sequences which retrieve and store $f_j(x)$.

Note that when a pointer x becomes inaccessible, the block hb to which it points becomes recognisably subject to garbage collection.

To illustrate the possible effect of the type of optimisation which has just been described, we consider the Huffman tree algorithm *huftables* of O.P.II., p. 149. This contains only one call to newat; the blank atom *n* created by this call appears as argument in three functions *l*, *r*, and *wfreq*. These functions satisfy conditions (a-d) above; hence it is seen that *n* can be created as a pointer to a block of three components. This leads us directly to a list-like representation of the tree implicit in the *huftables* algorithm, and hence to an automatic implementation of this algorithm rather like that which might be used if the algorithm were implemented manually.

It may well be possible to extend the preceeding considerations to multi-argument mappings one of whose parameters is a blank atom. However, in the present newsletter we shall not pursue this line of thought.

9. Allocation of Objects within Areas.

The fact that SETL uses garbage collection imposes a substantial overhead on SETL programs. In languages like PL/1 which allow objects to be allocated within areas which can be freed explicitly, much of this overhead is avoided, essentially because space recovery can be carried out more quickly when an entire area is freed than when items must be individually classified live/dead by a garbage collector. In the present section we shall sketch a method which may allow an optimiser to find effective schemes for allocating SETL objects within areas, and also to discover program points at which these areas can be freed.

Our idea is as follows: suppose that an object V is created when the source expression of an ovariable o is evaluated. Consider a program point p_o not lying along an edge belonging to the set

(1) $$\text{liveval}(o) = [+:i \in \text{crthis}^{-1}(o)]\ \text{pcrthis}(i,o).$$

Then the object V will never be used after p_o is passed. Hence, if we use an area A to allocate space for storage of each value V created at o, we will be in a position to insert an instruction which frees A when p_o is reached. However, to free A we must be sure that no object in A is a member or a component of any object s not in A, since if this condition were not satisfied even tests like $s \underline{\text{eq}} s_o$ would fail after A were freed.

To carry out the analysis which these considerations suggest, we construct a graph G whose nodes are the ovariables o of a program P to be analysed. Given two ovariables o and o_1, we draw a directed edge from o_1 to o if o ε crmemb(o_1), or o ε crcomp(o_1,n) for some n, or o ε crsomcomp(o_1). Strongly connected regions of this graph represent groups of ovariables whose values should be allocated within the same area since they must always be freed together. For this reason, we reduce each strongly connected part R of G to a point by identifying the members of such an R. This transforms G into a loop-free graph G'. Given an n ε G, we find the set *pred(n)* of all of the predecessor nodes n_1 of n in G', and the set *ovars(n)* of all ovariables o which are members of an n_1 belonging to pred(n). The potential *freeing points* for n are the points x in the flow graph of p which belong to the intersection

(2) freepoints(n) = [*: o ε ovars(n)] killsval(o),

where *killsval(o)* is the set of edges complementary to the set *liveval(o)* of (1). We introduce an area A(n) for each n ε G'. However, if two nodes n_1, n_2 ε G_1' have the same set of freeing points, the same area should be used to allocate space for the ovariables both of ovars(n_1) and of ovars(n_2); hence A(n_1) and A(n_2) may as well be identified. Moreover, if n_3 ε G has the property that freepoints(n_3) ⊆ freepoints(n_1) while every other n ε G for which freepoints(n) ⊆freepoints(n_1) satisfies freepoints(n_3)⊆ freepoints(n), then there will never be reason to free A(n_3) without freeing A(n_1), and A(n_3) can be allocated as a subarea of A(n_3). Whenever we insert an instruction freeing A(n) we must also insert an instruction freeing A(n') whenever n' is a predecessor of n in G'; unless of course, A(n') is a subarea of A(n) or is identical to A(n). Note also that the rule just stated will introduce redundant deallocation operations; however, these can readily be removed using an ordinary redundant-operation removal technique.

If o ε ovars(n), and if the operation defining the value of o will build up a SETL object for which space must be allocated, the necessary space should be allocated in the area A(n).

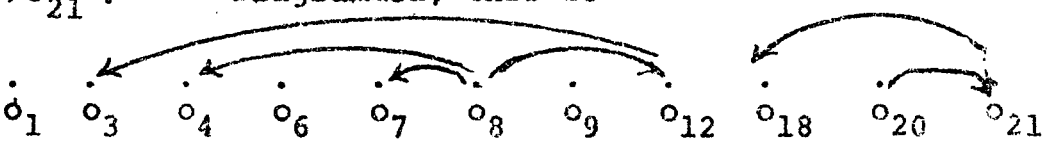
To illustrate the preceeding considerations we consider the *path* algorithm of O.P.II, pp. 123-124. We write this code in a form more conventent for our purposes as follows:

```
/* x, y, gr, f, and cap are defined before the following code
                                                    is entered */

o1:  new = {y};
o2:  set = new;
o3:  next = nl;
(while  new ne nl)
o4:         newer = nl;
o5:         (∀v ε new)
o6:         grofv = gr{v};
o7:         prior = nl;
o8:                  (∀uε grofv)
o9:                           capquant = cap (<u,v>,f,c)
                              if u ε set or capquant gt o then continue;;
o10:                          prior = prior + {u};
                     end ∀u;
o11:                 (∀uε prior)
o12:                          pair = <u,v>;
o13:                          next = next + {pair};
                              if u eq x then go to done;;
```

```
o14:                        set = set + {u};
o15:                        newer = newer + {u};
                        end ∀u;
                    end ∀v;
o16:                new = newer;
                end while;
o17:            pth = Ω;
                go to finish;
done:
o18:            pth = nl;
o19:            pt = x;
                (while t)
o20:                    nextpt = next(pt);
                        if nextpt eq Ω then quit;;
o21:                    pair = <pt, nextpt>;
o22:                    pth = pth + {pair};
o23:                    pt = nextpt;
                end while;

finish: /* after this point, the variables new, set, newer, graft,
           prior, capquant, pair, next, u, v, and nextpt are
           all dead.  However, pth remains alive */
```

The allocation routines which have been outlined would analyse this code approximately as follows: the operations defining o_2, o_{10}, o_{13}, o_{14}, o_{15}, o_{16}, o_{17}, o_{19}, o_{22}, and o_{23} either simply transmit pointers or use them destructively, and for this reason are ignored.

The loophead operations o_5 and o_{11} are basically extraction operators, and hence may be assumed not to create any new objects. Thus the ovariables at which space must be allocated are o_1, o_3, o_4, o_6, o_7, o_8, o_9, o_{12}, o_{18}, o_{20}, and o_{21}. Let us write $o \longrightarrow o'$ to indicate that an edge runs from o to o' in the graph G introduced above. The only edges present if such a graph is formed, for the code shown above are $o_{12} \longrightarrow o_3$, $o_8 \longrightarrow o_4$, $o_8 \longrightarrow o_7$, $o_8 \longrightarrow o_{12}$, $o_{21} \longrightarrow o_{18}$, and $o_{20} \longrightarrow o_{21}$. Diagrammed, this is



This graph is clearly of a very simple structure; indeed, it is largely disconnected. Freeing points are shown in the following table:

ovariable	freeing all points	irredundant freeing points
o_1	o_{16} - finish	$o_{16}(+)$
o_3	o_1 - o_2, finish	finish
o_4	o_1 - o_3, o_{17} - finish	o_{17}
o_6	o_1 - o_5, o_{11} - finish	o_{11}
o_7	o_1 - o_6, o_{16} - finish	$o_{16}(-)$
o_8	o_1 - o_2, finish	finish
o_{12}	o_1 - o_2, finish	finish
o_{18}	none	———
o_{20}	none	———
o_{21}	none	———

Thus our analysis leads us to introduce five areas, which we may designate as A(1), A(2), A(3), A(4), and A(5). These may be allocated within each other, in the pattern A(1) $\subseteq$ A(2) $\subseteq$ A(3) $\subseteq$ A(4) $\subseteq$ A(5). The set *grofv* is formed within A(1), which is freed immediately after the end of the ($\forall$u$\in$ grofv) - loop. The set *prior* is formed within A(2), which is freed immediately after the end of the ($\forall$u$\in$ prior) - loop. The set {*y*} is formed within A(3), which is freed immediately after o_{16}. (Since this set consists only of a single element, it is of course better to identify A(3) with A(4) and to suppress this freeing operation). The set *newer* is formed in A(4), which is freed on exit from the first *while* loop of the above code. The set *next*, as well as the items *pair* and *u*, are formed in A(5) and freed on exit from the above routine. All of this describes a reasonably acceptable allocation/freeing scheme. Of course, the scheme obtained is one which aims singlemindedly to free areas as soon as possible. A more sophisticated analysis would attempt to move deallocation operations out of loops, and might be able to trade garbage-collector time for other forms of execution time in a more sensible way.

10. Detecting 'Stack' objects.

Space for storage of SETL objects is ordinarily reserved within a garbage-collected 'heap' area; and a 'stack' area is used for recursion control. However, since space is more efficiently recovered from the stack than from the heap, there is advantage to be gained by allocating storage blocks on the stack rather than on the heap when this is possible; a block allocated on the stack is recovered simply by dropping the 'stack top' pointer below the start of the block. In the present section, we shall describe conditions under which this can be done (automatically, by a SETL optimising compiler.)

Objects allocated on a stack during the running of programs P are normally associated with 'blocks' within P, in the sense that these objects remain 'allocated' only within the block, and are automatically deallocated (by restoring the 'stack top' pointer to a prior value) when exit is made from the block. The 'begin' blocks of ALGOL 60, PL/1, and BALM all play this role. In SETL,blocks having this semantic function are not specifically defined, and we may therefore ask 'what program subparts are to play the role of these "blocks"?' We might choose to have subprocedures play this role; alternatively, program intervals might be chosen. We shall suppose in this section that intervals are to play the role of blocks. An object x defined by an ovariable o appearing within an interval I can be allocated on the stack and deallocated on exit from I provided that the following conditions are satisfied:

i. The object x is alive only within I; moreover, it is not made a part of any composite object which remains alive outside of I.

ii. The object x is never used destructively to produce an object which either remains alive outside I or is made part of a composite object which remains alive outside I.

'Stack objects' can easily be distinguished once the auxiliary functions described in section 2 and 4 have been calculated. We proceed as follows:

a) First build the set Δ of all those ovariables of I which have no uses outside I. Using the inverse of the function *erpart* of section 2, select a subset Δ' of Δ by eliminating all those variables which belong to *erpart*(*i*) for some ivariable i not occuring in I.

b) When a pattern of destructive uses has been decided on, check the ovariable o' of each operation in I containing a destructive ivariable use, to ensure that o' belongs to Δ'. If not, let i' be the ivariable of o' which is used destructively, and find all ovariables $o \in \Delta'$ which belong to *erpart*(*i'*). Eliminate these o from Δ'. Repeat as long as Δ' keeps diminishing. The o which remain in Δ' when it stops diminishing define the values which may be formed on the stack rather than in the heap.

If the technique just outlined is used, one will probably want to distinguish between the part of the stack which contains the root words of 'stack variables' and the part which contains object representations. This can be done easily by using appropriate pointers, which are themselves stacked and unstacked as one enters and exits from intervals; some modification of the present SETL garbage collector, and especially of its 'move' portions, may also be required.

11. Appendix. Pointer overlap in PL/1.

Methods of analysis like those described in section 2 and 6 of the present newsletter can be applied to the analysis of *pointer overlap* in PL/1. Two pointer-variables p_1 and p_2 in a PL/1 program P are said to overlap if they can possibly point to the same run-time object.

If P is not analysed in some helpful way for pointer overlap, then in compilers P it may be necessary to make the 'worst case' assumption that all pointers overlap, and so crude an assumption may conceal redundant expressions and opportunities for code motion that could otherwise be found.

In order to keep our discussion short, we shall analyse the pointer overlap problem under a number of simplifying assumptions. P is assumed to be schematised as a set of basic blocks, each of which consists of schematised instructions. We allow instructions of the following form:

(1)	single-variable allocation:		p = allocate(x)
	array allocation	:	p = allocate(A)
	based retrieval	:	$p = p_1 \rightarrow x$
	based indexed retrieval	:	$p = p_1 \rightarrow x(n)$
	based storing	:	$p \rightarrow x = p_1$
	based indexed storing	:	$p \rightarrow x(n) = p_1$

In addition, we allow arithmetic, boolean, etc. operations which do not create, fetch, or store a pointer value. Note that in confining our list of pointer operations to (1), we ignore the complications which arise from the existence of an ADDR function. The details needed to treat structures are also ignored, as is the treatment of non-based pointer variables and pointer arrays. Note that operations known to be arithmetic will be ignored in the analysis which follows; in particular, we shall concern ourselves with indexed retrievals $p = p_1 \rightarrow x(n)$ and indexed storage operations $p \rightarrow x(n) = p_1$ only when x is an array of pointers. These omissions in our present discussion are easily made up; note however that use of the ADDR function will blur the results of our analysis badly.

We introduce the predicates *alloc(a)*, *retr(a)*, *stor(a)* to describe the classes of operators appearing in (1) above.

We introduce functions analogous to those used in section 2. Let o be an ovariable('output variable' or 'definition') and i an ivariable ('input variable' or 'use') of P. Then

a) *crthisp(o)* denotes the set of all o_1 which create a pointer which might become the value of o;

b) *crmembp(o)* denotes the set of all o_1 which create a pointer which might be stored into some value or array at which some value of o can point.

Similar functions are introduced for ivariables i. Let ud(i) be the set of all output variables chained to i (by the usual use-definition chaining process). Then plainly

(2)
```
crthisp(i) = [+: oε ud(i)] crthisp(o);
crmembp(i) = [+: oε ud(i)] crmembp(o);
```

The corresponding equations for ovariables are somewhat more complicated. For an allocation operation o, we have crthisp(o) = {o} while crmembp(o) is null. For a based or based indexed retrieval with p_1 as input pointer, we have crthisp(o) = crmembp(p_1). To calculate crmembp(o) when retr(o), a more complex relationship must be used. Let p_1 appear as input pointer in the operation defining o. Then any o'ε crmembp(p_1) can be a value of $p_1 \rightarrow x$; and crmembp(o) may be estimated (from above) as the set of all pointers $\bar{v}$ at which the pointer v' created by such an o' may point. But for v' to point at $\bar{v}$, there must exist an operation $p' \rightarrow x' = \bar{p}$ for which v' is a possible value of p' and for which $\bar{v}$ is a possible value of $\bar{p}$. Thus for $\bar{o}$ to belong to crmembp(o) there must exist o'ε crmembp(p_1) and an instruction $p' \rightarrow x' = \bar{p}$ with o'ε crthisp(p') and $\bar{o}$ ε crthisp($\bar{p}$). Let us write *crthisp*$^{-1}$ for the mapping from ovariables to ivariables which is inverse of the mapping *crthisp*. Let *usfirst*(i) denote the proposition 'ivariable i occurs as the first argument of a store operation';

let lastarg(i) map each i for which isfirstinstor(i) into the last argument of the operation in which i occurs. Then the argument that has just been given shows that when o is the output variable either of $p = p_1 \longrightarrow x$ or of the corresponding indexed retrieval operator, we can estimate crmemb(o) as

(3) crmemb(o) = crthisp [{lastarg(i), i ∈ crthisp^{-1} [crmembp(arg1(o))] |isfirstinstor(i)}].

For a based storing operator $p \longrightarrow x = p_1$, we count p as ovariable, and both p and p_1 as ivariables; then crthisp(o) = crthisp(p) (representing no change) and crmembp(o) = crmembp(p) + crthisp(p_1). The same rule applies to based indexed string operators.

We have agreed to write alloc(o), retr(o), and stor(o) if the operation defining o is of allocation, retrieval, or storage type (indexed or unindexed) respectively. Let arg1(o) denote the p_1 of (1), and, if stor(o), let arg0(o) denote the p appearing in (the fifth and sixth lines of) (1). Then by what has just been said and using the notations just introduced, we may write the equations

(4)
```
crthisp(o) = if alloc(o) then {o}
             else if retr(o)then crmembp(arg1(o))
             else /* if stor(o) then */ crthisp(arg0(o));
crmembp(o) = if alloc(o) then nl
             else if retr(o) then
crthisp[{lastarg(i),i ∈ crthisp^-1 [crmemb(arg1(o))
                                   |isfirstinstor(i)}]
             else /* if stor(o) then */ crmembp(arg0(o))
                                        + crthisp(arg2(o));
```

These equations may be solved by a suitable variant of the general 'monotone convergence' method described in section 2