SETL Newsletter # 134

J. Schwartz
July 1, 1974

Inter-Procedural Optimisation

The optimisation algorithms described in newsletter 118, 130, 131 and A. Tenenbaum's thesis are all intraprocedural, in that they work with a schematised program consisting of one single 'main routine' which is assumed to be free of subprocedure calls. These algorithms also ignore the complications to flow analysis which arise when subprocedures become the values of variables or of parameters, and when transfers to variable labels occur within a program. A useful optimiser system will of course have to handle both these situations. In the present newsletter we shall outline algorithms, adapted from those developed for a PL/1 optimiser by F. Allen and her associates, which begin to be adequate to this purpose. It will be seen that in the form in which they will be presented these algorithms use some of the ideas developed in newsletter 131. We shall also employ the terminology introduced in that newsletter.

1. Flow Estimation, the Call Graph.

Let P be a program in which procedure and/or label variables are used. The first problem to be overcome is that when we encounter P we do not know what structure its flow, i.e. its program graph, has. This problem is particularly acute in SETL, since variables are untyped, so that any variable can take on a procedure value, and since the map applications f(x) and $f(x_1,\ldots,x_n)$ can also be function calls. Before applying other global optimisations we must therefore aim to estimate P's flow, i.e. to determine which variables v in P can have procedure (resp. label) values, and the procedures (resp. labels) which can become values of each such v.

This information must be developed by an algorithm which does not require precise information concerning P's flow, since it is this flow which we are trying to find. Note that the algorithms in which subsequently use the flow will remain correct even if the flow is overestimated, but not if is is underestimated. Here we speak of an *overestimation of flow* if a transfer or call that is actually impossible is judged possible, and of an *underestimation* if a possible transfer or call is judged impossible. We may speak in a similar sense of overestimation (and underestimation) of data flow: data flow is overestimated if we adjudge the set ud(i) of all ovariables o which can set the value of a given ivariable i to be larger than it is.

To estimate ud(i), we begin with a coarse overestimate $ud_o(i)$ and refine it. The initial guess $ud_o(i)$ is obtained as follows. P is schematised in a manner 'resolving' the names used within P, i.e., assigning explicitly different symbols to names which have the same spelling but which appear in different namescopes. Then we put the ovariable o in $ud_o(i)$ if i and o involve the same symbol x (i as a use, o as a definition). This rule applies if neither o nor i is a subprocedure parameter or argument. Subprocedure arguments are treated both as ivariables and ovariables, rather as if a function call of the syntactic form $y = f(a_1,\ldots,a_n)$ consisted of two successive statements

(1)
$$\underline{\text{enter}}\ f(a_1,\ldots,a_n)$$
$$y = f(a_1,\ldots,a_n);$$

with $a_1,\ldots,a_n$ in the first line being ivariables and $a_1,\ldots,a_n$ in the second line being ovariables. (The reader is reminded that a SETL subprocedure can modify the parameters with which it is called, and that a subroutine can be regarded as a function which returns a nil value to a dummy variable that is never used.)

A procedure parameter appearing in a header line

(2) define sub(p_1,...,p_n)

or

(2') definef f(p_1,...,p_n)

is considered to be an ovariable, which receives a value from the corresponding argument when the subprocedure is entered. In setting up $ud_o(i)$, we use the following rules to handle subprocedure arguments and parameters. For the ivariable a_j appearing in the first line of (1), $ud_o(a_j)$ includes each ovariable p_j appearing in a header (2) or (2') for which n = m. Variables appearing in the context

(3) return v

within a function headed by (2') are ivariables, and $ud_o(v)$ includes each ovariable y appearing in the second line of the 'expanded form'(1) of a call. Moreover, a return statement of the form (3) or of the simpler form

(3') return

used within a subroutine is considered to have every parameter p_j of the subroutine in which it appears as an ivariable; and the ovariable a_j appearing in the second line of (1) belongs to $o(p_j)$ whenever the call (1) and the subroutine header (2) have a matching number of parameters.

The function *f* (or the subroutine *sub*) is considered to be an ovariable of the header line (2') or (or (2)) which defines *f* and thus acts very much like an assignment to *f*.

In this same sense, we consider a labeled statement

(4) lab: ...

to have the symbol *lab* as an ovariable, since (4) defines *lab* and thus acts very much like a assignment to *lab*.

The rules stated above define $ud_o(i)$ for each ivariable of P. To refine this overestimate of the data flow, we proceed as follows. Using ud_o in place of the use-definition chaining function ud which appears in newsletter 131, equations (1,2) of section 2 and equations (1-4) of section 4, we calculate the functions *crthis*, *crmemb*, *crcomp*, *crsomcomp* appearing in these equations. The reader is reminded that *crthis(i)* (resp. *crthis(o)*) is the set of all ovariables which create an object which at some moment in the execution of P becomes the current value of i (resp. o). Moreover, *crmemb(i)* (resp. *crsomcomp(i)* is the set of all ivariables *j* whose values become incorporated as members into a set (resp. as components into a tuple) which at some moment in the execution of P becomes the current value of *i*; *crmemb(o)* and *crsomcomp(o)* can be defined in a very similar way for ovariables o. Note that the function that we primirily want is *crthis(i)*, and that *crmemb(i)*, etc. are merely needed as auxiliary quantities for calculating *crthis*.

Let o designate the ovariable of a subprocedure header-line (2) or (2'), in the sense explaned in the sentence preceeding (4) above. The subprocedure declared by (2) or (2') can become the value of an ivariable i only if $o \in crthis(i)$. Similarly, of o' designates the label-representing ovariable of the labeled statement (4), then an ivariable i can have the *lab* of (4) as one of its values only if $o' \in crthis(i)$.

Thus, by calculating *crthis* we obtain conditions, which should be reasonably accurate, restricting the variables which can assume procedure and label values, as well as the particular procedure and label values which these variables can assume. These conditions give us our first estimate (actually, overestimate) of the flow of P.

Note in passing that since the number of procedures and explicit labels appearing in a SETL program P will generally be far smaller than the number of ivariables of P, it may be preferable to calculate not *crthis(i)* but its inverse function $crthis^{-1}(o)$. Equations somewhat like the equations given in section 2 and 4 of newsletter 131 but involving these inverse functions can be written and actually will be outlined in later section of the present newsletter.

Once having arrived at a first estimate of the flow of P, we may revise our preliminary overestimate $ud_0(i)$ of the data-flow of P, obtaining a more precise overestimate $ud_1(i)$. Using ud_1 in place of ud_0 and recalculating $crthis^{-1}(i)$ by the method just explained, we can obtain a still more precise estimate $ud_2(i)$ and so on inductively. Since all the sets involved are finite, this process will stabilise after a finite number of steps. Actually, since each such iteration may be relatively expensive, we may decide to iterate only once or twice, and to use the flow estimate corresponding to $ud_0(i)$ or $ud_1(i)$ in applying other global optimisation processes to P.

To calculate ud_{n+1} from the flow F_n estimated from ud_n, we could proceed as follows. In very much the ordinary way, P is schematised into a collection of basic blocks. When a block B ends with a transfer

(5) go to *labvar*

where *labvar* is a variable having the labels $L_1,\ldots L_n$ as possible values, then B has as sucessors the blocks $B_1,\ldots,B_n$ beginning with $L_1,\ldots,L_n$ respectively. Points at which functions of the form $y = f(a_1,\ldots,a_n)$ might be called are expanded as follows. Let the procedures which can be a value of f be called $rout_1,\ldots,rout_k$. Let $p_{j1}, p_{j2},\ldots, p_{jn}$ be the formal parameters of $rout_j$, and let p_{jo} denote the value returned by $rout_j$. Let a label r_j be generated to mark the point of entry to $rout_j$. Then the function call $y = f(a_1,\ldots a_n)$ is expanded into the following code schema.

(6)

```
        go to labvar;

        /* where the possible values of labvar
           are the generated labels ℓ1,ℓ2,...ℓn
           appearing immediately below */

ℓ1: <p11, p12,...,p1n> = <a1,...,an>;
    /* pj1,...,pjn are the parameters of routj */
      go to r1; /* rj is the point of entry to routj */
retlab1: <a1,...,an> = <p11,p12,...,pin>;
y = p10; /* pjo designates the value returned by routj*/
go to ℓ0;
ℓ2: <p21,p22,...,p2n> = <a1,...,an>;
    go to r2;
retlab2: <a1,...,an> = <p21,p22,...,p2n>;
        y = p20;
        go to ℓ0;
        ...
```

```
lk: <p_k1, p_k2,...,p_kn> = <a_1,...,a_n>;
    go to rk;
retlabk: <a_1,...,a_n> = <p_k1,p_k2,...,p_kn>;
    y = p_ko;
l0: /* no-operation; but marks the end of the
                        preceeding code sequence */
```

A statement

(7) return p_o;

occuring in the routine $rout_j$ is treated as if it were a transfer (5) for which the possible values of *labvar* are the collection of all generated return labels *retlabi* appearing in an expansion of the form (1) immediately prior to a transfer to the point of entry to $rout_j$.

Because of the way that subprocedure calls are handled in (6), loops containing calls will appear as multi-entry loops, and the calculation of ud_{n+1} from the flow estimate determined by ud_n may involve considerable node splitting. Moreover, the subroutine treatment which has been outlined loses sight of the fact that a procedure called from one place cannot return as if it had been called from another. We conclude that this simple scheme for treating subprocedures overestimates the flow of P in a significant way, and that it should be replaced by a treatment which is more precise and which makes repeated node splitting unnecessary. An algorithm for calculating *ud*, modified to meet these objections, will be sketched in the next section of the present newsletter.

The analysis that has been described in the preceeding pages allows us to detect situations in which a procedure is called with the wrong number of arguments.

These will appear as situations in which a subroutine *rout* with n parameters is transmitted as value to a procedure call $f(a_1,...,a_n)$ requiring a number of parameters $m \neq n$. If f is seen to have no possible value other than *rout*, a fatal diagnostic can be issued; in other cases, a nonfatal warning is more appropriate. By making 'subroutine of n parameters' a type in the calculas of types described by A. Tenenbaum additional information concerning procedures called with the wrong number of parameters can be uncovered by the type-finding process.

Note that the analysis which has been described also uncovers the *call graph* G of the program P. This is the graph whose nodes are the subprocedures s_1, s_2,... P, and in which s_2 is a sucessor of s_1 if and only if s_1 contains a call to s_1. A set of subprocedures belonging to a strongly connected region of G are said to be *corecursive* with each other, and the procedures belonging to them are said to be recursive. The information made manifest in G is useful for various optimisations, e.g. non-recursive routines can use simplified linkage convertions.

2. Calculation of the data flow mapping ud(i).

In the present section, we assume that the control flow of P has been estimated, i.e. that we have found all variables which can assume label or procedure values, and have estimated the label and procedure values which each such variable can assume. We will describe a procedure for calculating ud(i) and other important data-flow related mappings from this estimate. This procedure, largely taken from recent work of F. Allen, treats subroutines in a special way which avoids the objections noted in the preceeding section to the subroutine treatment outlined there.

Our aim is to treat procedure calls as elementary operations rather than as transfers which complicate the flow of P. To treat a call *q* to a subroutine *sr* as elementary, we must ascribe three values to it:

i. The collection *defsof(q)* of all definitions (ovariables) occuring in *sr* which might supply the value of a variable used immediately after return from *q*;

ii. the collection *usesin(q)* of all variable-uses (ivariables) occuring in *sr* which use the value which the corresponding variable has immediately before *sr* is entered.

iii. The collection *thru(q)* of all variables v which can be transmitted thru *sr* along some path clear of redefinitions of v.

These functions are used to calculate various data-flow related mappings, among them the set ρ(b) of all definitions d which can reach the entrance to a block b along some path.

The basic relationship used for calculating this function is simply

(1) $\rho(q) = [+: p \in pred(b)]\ (\rho(p) * thru(p) + defsof(p))$,

where *pred(b)* is the set of predecessor blocks of b. The relationships (1) constitute a system of equations whose solution can in simple cases be obtained efficiently using, e.g., the interval method.

To handle a program P containing subroutines and subroutine calls, we proceed as follows. The flow of P is estimated and the call graph G of P determined. As already noted, any set of subprocedures represented by nodes of G belonging to a strongly connected subregion is said to be *corecursive*. If there exist corecursive subprocedures, we choose a non-null set B of edges of G such that G - B contains no cycles; B should be chosen so as to be minimal in some appropriate sense. Each edge belonging to B represents one or more calls from one procedure p_1 to another procedure p_2. For each p_2 appearing as the terminal node of such an edge e, we define a formal auxiliary routine p_2', and replace the call from p_1 to p_2 which e represents by corresponding call from p_1 to p_2. We then use an initial overestimate of *thru(p')* and *defsof(p')* for each of the auxiliary subprocedures p' introduced in this way, taking *thru(p')* to consist of all variables and *defsof(p')* to consist of all global variables or parameters of p which appear on the left-hand side of some assignment belonging either to p or to any procedure which might be called, directly or indirectly, from p. With B removed, the subgraph G - B of G is free of cycles i.e. is a tree; and by arranging this tree in a linear order we succeed in arranging all of the subprocedures represented by G in an order in which each procedure follows the procedures

(not in the set of formal auxiliary routines p') which it calls. Using Allen's term, we call this order the *inverse invokation order*; and this is the order in which we will process the collection of procedures constituting P. To process procedure *sr*, we take the set of all the global variables appearing in *sr* and in all the routines it calls (which are prior to it in inverse invokation order), and to this set append the formal parameters of *sr*, obtaining a set V. Then, to regularise the processing which will follow, we set up a block of formal assignments, one assignment for each of the variables in V, and prefix this block to the first statement of *sr*. The collection of ovariables of this block will be called the *external ovariables* of *sr* and will be designated by the symbol EXOV.

Next, we decompose *sr* into intervals, splitting nodes if necessary, and use equation (1) to calculate ρ(q) for each of the blocks q of *sr*. The values *thru(p)* and *defsof(p)* which we use in doing this are defined as follows:

a. For a basic block p, *thru(p)* is the intersection of the sets *thru(x)* associated with each of the individual statements x of p. If x is an assignment statement, then *thru(x)* consists of all variables other than the target variable of x. If x is a call to a subprocedure *ssr*, then *thru(x)* consists of all variables v which are not substituted for parameters p of the call x and which belong to *thru(ssr)*, plus those variables which are substituted for some parameter of *ssr* which belongs to *thru(ssr)*. If x is a call to a subprocedure which is somewhat indeterminate and might be either ssr_1, ssr_2,..., then *thru(x)* consists of all variables v for which there exists some j such that *ssr* = ssr_j satisfies the condition stated in the preceeding sentence.

Note that since all routines called from *sr* preceed *sr* in inverse invokation order, the value *thru(ssr)* will always be available during the computation that we have just described.

b. Let p be a basic block, x a statement of p, and o an ovariable of x. Then o belongs to *defsof(p)* if it belongs both to *defsof(x)* and to *thru(y)* for each y in p which follows x in the serial order of p. If x is an assignment statement, then *defsof(x)* is the target variable of x. If x is a call to a subprocedure *ssr*, then *defsof(x)* consists of all variables v which belong to *defsof(ssr)*, plus all variables v which are substituted for some parameter of *ssr* which belongs to *defsof(ssr)*. If x is a call to a subprocedure which is somewhat indeterminate and might be either ssr_1, ssr_2,..., then *defsof(x)* consists of all variables v for which there exists some j such that *ssr* = ssr_j satisfies the condition stated in the preceeding sentence.

Once ρ(b) has been determined by using these conventions and by making appropriate use of equation (1), we calculate

(2) [+: bε returnstats] ρ(b) * EXOV,

where *returnstats* is the set of blocks of *sr* which consist only of return statements (for convenience, each return statement is segregated into a block of its own.) Then *thru(sr)* is defined as the set of variables in V whose corresponding ovariables belong to the set (2). In addition, *defsof(sr)* is defined as the set

(3) [+: bε returnstats] ρ(b) - EXOV.

Note that the sets thru(p) and *defsof(p)* which result from our calculation satisfy *thru(p)* $\subseteq$ *thru(p')* and *defsof(p)* $\subseteq$ *defsof(p')* for each subprocedure p which is a terminal node of some edge of the subset B of the call graph G. The sets *thru(p')* and *defsof(p')* overestimate the true data flow functions which should be associated with the subprocedure p; the sets *thru(p)* and *defsof(p)* also overestimate these functions, but not as badly. By replacing *thru(p')* and *defsof(p')* by *thru(p)* and *defsof(p)* respectively and by repeating the computation which has just been described, we can improve our estimates. This improvement can be iterated as often as desired, in principle even until stable estimates of the functions *thru* and *defsof* result. But the high cost of iteration may only justify one or two iterations, which anyhow should in most cases yield reasonable estimates of *thru* and *defsof*.

Note that the function *defsof(p)* which the preceeding calculation associates with a subprocedure *p* tells us what assignment operations and procedure calls *c* internal to *p* can set the value of a global variable or parameter *v* when p itself is called. In some applications we will wish to obtain additional information, relating variables *v* not merely to the procedure calls *c* which set their variables, but to the specific elementary operations which assign values to *v* from within the routine called by *c*. This may for example be convenient if we intended either to propagate constants or to carry out one of the constant-propagation-like optimisation processes described in Newsletters 130 and 131. To obtain this additional information, we have only to note the variable or parameter v in which we are interested, and, using the value *defsof(c)*, find all the operations within the procedure called by *c* which set v; proceeding in this way and iterating through sucessive subprocedure call operations until a full transitive closure is formed, we will succeed in forming the set *defsofc(p)* of all assignment statements which can establish the values of v.

Then the set *defsofc(p)* rather than the set *defsof(p)* can be taken as an expression of the use-definition chaining relationships in the program being analysed. An alternative technique is to use *defsof(p)*, but to chain each variable in *defsof(p)* back to a nominal assignment v = v inserted immediately prior to each return statement occuring in the subprocedure p. Both of these techniques are valid for entirely general collections of mutually corecursive procedures.

For the'definition-to-use' part of the data-flow analysis process we will often need to have available a function *usesin(q)* which maps each subprocedure q into the set of all global variable or parameter uses occuring in q. The set *usesin(q)* is obtained by summing, over all the blocks b of q, all uses of variables belonging to ρ(b)*EXOV. The rule for associating a set of uses with a block b is as follows: Let x be a statement of b and i an ivariable of x. Then i belongs to the set *usesin(b)* if it belongs both to *usesin(x)* and to *thru(y)* for each y in b which preceeds x in the serial order of b. If y is an assignment statement, then i belongs to *usesin(y)* if it is an argument of the operation appearing on the right-hand side of y. If y is a call to a subprocedure *ssr*, then *usesin(y)* consists of all ivariables which either belong to *usesin(ssr)* or are substituted for a parameter belonging to *usesin(ssr)*. If y is a call to a subprocedure which is somewhat indeterminate and might be either ssr_1, ssr_2,..., then *usesin(y)* consists of all ivariables which either belong to some one of the sets *usesin*(ssr_2) or are substituted for a parameter belonging to some one of these sets.

A transitive closure technique like that described two paragraphs above may be used to chain definitions to the uses the values they define, even when theses uses are reached through a lengthy sequence of possibly recursive procedure calls.

Note once more that an estimate of the data flow within a program P leads to an estimate of the control flow within P, which in turn leads via the processes described in the present section to an improved estimate of data flow. Thus the entire data and control-flow estimation process that has been described can be iterated. However, it is unlikely that more than a very few iterations will either be feasible or required.

3. A few remarks on subroutine linkage optimisation.

Once the call graph of a program P has been determined and the other analyses described in the preceeding pages have been carried out, a number of small but useful subroutine linkage optimisations become possible.

i. Arguments which are never read by a subprocedure *sr* (i.e., 'output' arguments) need not be transmitted when *sr* is called; arguments which are never modified by *sr* need not be returned when return is made from *sr*.

ii. A function-type subprocedure *sr* is said to be *without side effects* if *sr* modifies none of its parameters, modifies no global variable, and if every variable local to *sr* is dead on entry to *sr*. Once *sr* is known to be without side effects, expressions containing calls to *sr* can be optimised in ways that would be impossible if *sr* had side effects. For example, redundant calculation elimination can be applied to such expressions.

iii. A subprocedure *sr* is said to be *non-recursive* if *sr* is not part of any cycle in the call graph G of the program P. (Note that non-recursive subprocedures can call recursive subprocedures and vice-versa.) Arguments to non-recursive subprocedures need not be transmitted via a system stack, but can be transmitted in a somewhat more efficient manner,

by direct assignment to an argument area associated with *sr* . Moreover, *sr's* temporaries need never be stacked. This saves stack manipulation on entry to and return from *sr*, allowing relatively compact and efficient call and return sequences to be used.

4. Equations for the inverse function $crthis^{-1}$.

We noted in section 1 that, since the number of procedures and explicit labels appearing in a SETL program P will generally be far smaller than the number of ivariables of P, it may be preferable to calculate not the function *crthis(i)* which maps each ivariable of P into the set of all ovariables which can create an object which becomes the value of i, but rather to calculate the inverse function $crthis^{-1}(o)$ directly. In the present section, we will sketch a system of equations, dual to the equations for *crthis*, which make direct calculation of $crthis^{-1}$ possible. To this end, we introduce a number of auxiliary functions. By *iuses(o)* we designate the inverse of the function *crthis(i)*; *iuses(o)* may also be described as the set of ivariables i in which there appears as argument an object created by evaluating o; by *ouses(o)* we designate the set of all ovariables o' in which such an object reappears. By *iholds(o)* designate the collection of ivariables i in which there appears as argument a set containing (as one of its members) an object created in evaluating o; by *oholds(o)* we designate the collection of ovariables o' at which such a set can appear. By *isomcomp(o)* we designate the ivariables i in which there appears as argument a vector (of possibly unknown length) having as component (in unknown position) an object created in evaluating o; by *osomcomp(o)* we designate the collection of ovariables o' at which such a vector can appear. Next, let n be an integer.

By *icomp(o,n)* we designate the collection of ivariables i in which a vector of a known length at least equal to n, having as its n'th component an object created in evaluating o, appears as an argument; by *ocomp(o,n)* we designate the collection of ovariables o' at which such a vector can appear.

Rather than confronting the full zoo of primitive SETL operations all at once, we shall at first simplify our discussion by ignoring tuple operations, and by assuming that the only four set-theoretic operations which appear in our schematized programs P are s+t, s-t, {x}, and ∃s. (A similar procedure is used in Newsletter 131, and is justified there.) The operations occuring in P may then be classified as transfer, null, inclusion, extraction, data, setalgebraic, copy, and other (non-set) algebraic operations (see NL 131, p. 5), and to describe these respective operation classes we introduce predicates *transf(o)*, *null(o)*, *incl(o)*, *extr(o)*, *data(o)*, *setalg(o)*, *copyop(o)*, *other(o)*. The operation forms described by these predicates are

transf(o): o = i_1;
null(o) : o = nl;
incl(o) : o = {i_1};
extr(o) : o = ∃i_1;
data(o) : o = *data*;
setalg(o): (for sets)
setalgpls(o): o = i_1 + i_2;
setalgmns(o): o = i_1 - i_2;
copy(o) : o = copy(i_1)
other(o) : o = i_1 + i_2; o = i_1 - i_2, etc. (for atoms).

For the functions *iuses* and *iholds* we have the following equations:

(1) $$\text{iuses}(o) = [+: o' \in \text{ouses}(o)]\ du(o')$$
$$\text{iholds}(o) = [+: o' \in \text{oholds}(o)]\ du(o').$$

(Here, *du* is the definition-to-use chaining function provided by data-flow analysis; it chains each ovariable o to the set of all ivariables which can be reached from o along a path free of redefinitions of the variable appearing in o). The *ouses* function obeys a slightly more complex set of equations. If i is an ivariable, let *out(i)* designate the target ovariable of the schematised assignment statement in which i appears as an argument, and let *argpos(i)* denote the (numerical) argument position in which i appears. Then the value created by evaluating o can reappear either as the output of a transfer operation whose argument belongs to *iuses(o)*, or as the output of an extraction whose argument belongs to *iholds(o)*. Thus we have

(2) $$\text{ouses}(o) = \{\text{out}(i),\ i \in \text{iuses}(o) \mid \text{transf}(\text{out}(i))\} + \{\text{out}(i),\ i \in \text{iholds}(o) \mid \text{extr}(\text{out}(i))\}.$$

The equation for *oholds(o)* is substantially more complicated. A set s having among its members some object created by evaluating o can appear as the output of an operation of transfer, setalgebraic, or copy type, provided that an appropriate input argument of this operation belongs to *ihold(o)*. Moreover, s can appear as the output of an inclusion operation, provided that the input to this inclusion belongs to *iuses(o)*. Finally, s can appear as the output of an extraction operator $\bar{o} = \exists\, i$. For this to happen, i must belong to some set *oholds(o')* for which o' belongs to *oholds(o)*. In consequence of all these facts we have the following equation for *oholds(o)*:

```
(3)  oholds(o) = {out(i), iε iholds(o)|setalgpls(out(i)) or
                                       transf(out(i)) or copy(out(i))}
               + {out(i), iε iholds(o)|setalgmns(out(i)) and
                                                argpos(i) eq 1}
               + {out(i), iε iuses(o)| incl(out(i))}
               + [+: o' ε oholds(o)] {out(i), iε oholds(o')|
                                                extr(out(i))}.
```

The system of equations (1-3) can be solved by a straightforward monotone convergence procedure. Note that in applying these three equations to determine data flow in the presence of label and procedure variables, we would only calculate *oholds(o)* for the relatively small number of ovariables which define labels or procedures, and for any additional ovariables to which our attention is directed in the course of solving equations (1-3).

If we now pass to a discussion of full SETL by admitting the existence of tuple operations, the preceeding equations undergo substantial complication: tuple operations of tuple-former, component extractor, subtuple extractor, tail extractor, component insertion, and tuple concatenation type, which we designate by the predicates *tform(o)*, *compex(o)*, *subtex(o)*, *tailex(o)*, *inxa(o)*, and *concat(o)*, appear in our schematised programs P. These classes of operators have the following typical forms:

$$tform(o)\text{: } o = \langle i_1, \ldots, i_n \rangle$$

compex(o):	$o = i_1(i_2)$	(i_1 a tuple)
subtex(o):	$o = i_1(i_2{:}i_3)$	"
tailex(o):	$o = i_1(i_2{:})$	"
inxa(o) :	$o = [i_1(i_2) \leftarrow i_3]$	(o a tuple)
concat(o):	$o = i_1 + i_2$	(i_1, i_2 tuples).

Note that the component insertion operation, which for conformability with our general ovariable/ivariable conventions we shall write as $o = [i_1(i_2) \leftarrow i_3]$, is ordinarily writtwn as v(n) = c; v is both the o and the i_1 of our schematic convention. In what follows, we shall make use of functions *arg1(o)*, *arg2(o)*,etc. which extract the first, second, etc. components of the operator whose output ovariable is o. It is also convenient for us to make use of two auxiliary functions *ianycomp(o)* and *oanycomp(o)*. The set *ianycomp(o)* designates the collection of ivariables i in which there appears as argument a vector (of possibly unknown length) having as component (in a position which is either known or unknown) an object created in evaluating o; by *oanycomp(o)* we designate the collection of ovariables at which such a vector can appear. Note the distinction between *isomcomp(o)* and *ianycomp(o)* (and the parallel distinction between *osomcomp(o)* and *oanycomp(o)*): an ivariable belongs to *ianycomp(o)* if its value can be a vector in which a certain object appears in any component position whether known or unknown, but belongs to *isomcomp(o)* only if this object appears either in an unknown component position or in a vector whose length is unknown. Thus *ianycomp(o)* always includes *icomp(o,n)*, while *isomcomp(o)* need not include *icomp(o,n)*. As in Newsletter 131, we use functions *known(o)* and *known(i)*, which have the value Ω if o (resp.i) is either an integer of unknown value or a vector of unknown length, and have the value n if o(resp.i) is either an integer of value known at compile time to be n, or a vector of length known to be n.

Using these conventions, we may state the following revised equations for the functions *iuses*, *iholds*, *isomcomp*, *icomp*, *ianycomp*, *iuses*, *oholds*, *osomcomp*, *ocomp*, and *oanycomp*.

(4)
```
iuses(o) = [+: o' ε ouses(o)] du(o');
iholds(o) = [+: o' ε oholds(o)] du(o');
icomp(o,n) = [+: o' ε ocomp(o,n)]{i ε du(o')|known(i) ne Ω};
ianycomp(o) = [+: o' ε oanycomp(o)] du(o');
isomcomp(o) = [+: o' ε osomcomp(o)] du(o')+
+ [+: o' ε oanycomp(o)|(known(o') is k) ne Ω]
                [+: 1 ≤ n ≤ k |o' ε ocomp(o,n)]
                                    {i ε du(o')|known(i) eq Ω};
```

The equations for *ouses*, *oholds*, *ocomp*, *oanycomp*, and *osomcomp* can be written most easily if we introduce the following macro:

(5)
```
macro     againextractions(o);
     ({out(i), iε iholds(o) |extr(out(i))}
     +{out(i), iε isomcomp(o)|compex(out(i)) and
                                        argpos(i) eq 1}
     +{out(i1), i1 ε ianycomp(o)|compex(out(i1)) and
                                    argpos(i1) eq 1 and
if (known(arg2(out(i1))) is n) eq Ω then t
        else if icomp(o,n) eq Ω then f else i1 ε icomp(o,n))
endm  againextractions;  /* which marks the macro's end */
```

The set *againextractions(o)* is the set of all o' in which the object created by evaluating o reappears by extraction either of an element from a set or of a component from a tuple. Using this macro, we may write the following equations:

```
(6)     ouses(o) = {out(i),i∈iuses(o)|transf(out(i))}
                          + againextractions(o);

(7)     oholds(o) = {out(i), i∈iholds(o)|
                    setalgpls(out(i)) or transf(out(i)) or
                          copy(out(i))}
                   +{out(i), i∈iholds(o)|setalgmns(out(i)) and
                          argpos(i) eq 1}
                   +{out(i), i∈iuses(o)|incl(out(i))}
                   + [+: o'∈oholds(o)] againextractions(o');

(8)     ocomp(o,n) = /* here we assume known(o) ne Ω and
                      1 ≤ n ≤ known(o) */
        {out(i), i∈icomp(o,n)|transf(out(i)) or
            copy(out(i)) or(concat(out(i)) and
                          argpos(i) eq 1)}
        +{out(i), i∈iuses(o)|tform(out(i)) and argpos(i) eq n}
        +{out(i), i∈iuses(o)|inxa(out(i)) and argpos(i) eq 3
                          and known(arg2(out(i))) eq n}
        +{out(i1), i1∈icomp(o,n)|inxa(out(i1)) and
                          argpos(i1) eq 1 and
                           known(arg2(out(i1))) ne n}
        +{out(i1),i1∈ianycomp(o)|tailex(out(i1)) and
            argpos(i1) eq 1 and
                 i1∈(icomp(o,known(arg2(out(i1)))+n) orm nl)}
        +{out(i1), i∈ianycomp(o)|subtex(out(i1)) and
            argpos(i1) eq 1 and
                 i1∈(icomp(o,known(arg2(out(i1)))+n) orm nl}
        +{out(i2), i2∈ianycomp(o)|concat(out(i2))and argpos(i2)eq 2
       and  i2∈(icomp(o,n-known(arg1(out(i2))) orm nl)}
        + [+: o'∈ocomp(o,n)] againextractions(o');
```

```
(9)     oanycomp(o) =
        {out(i), i∈oanycomp(o)|transf(out(i)) or
             copy(out(i)) or concat(out(i)) or (tailex(out(i)  or
             subtex(out(i)) and argpos(i) eq 1) or
             (inxa  (out(i)) and argpos(i) eq 1))}
        +{out(i), i∈iuses(o)|tform(out(i1))or
             (inxa(out(i)) and argpos(i) eq 3)}
        + [+: o'∈oanycomp(o)] againextractions(o');


(10)    osomcomp(o) =
   {out(i), i∈isomcomp(o)|transf(out(i)) or copy(out(i)) o:
        concat(out(1)) or(tailex(out(i)) and argpos(i) eq ) or
             (subtex(out(i)) and argpos(i) eq 1)}
 + {out(i), i∈ianycomp(o)|concat(out(i)) and known(out(i)) eq Ω and
        if(known(i) is k) eq Ω then f else
                1 ≤ ∃n ≤ k | i∈(icomp(o,n) orm nℓ)}
 + {out(i), i∈ianycomp(o)|(tailex(out(i)) or
        subtex(out(i))) and argpos(i) eq 1 and
        known(out(i)) eq Ω and if(known(i) is k) eq Ω then f
         else 1 ≤ ∃n ≤ k|i∈icomp(o,n) orm nℓ}
 + {out(i), i∈iuses(o)|inxa(out(i)) and argpos(i) eq 3
        and(known(arg2(out(i))) eq Ω or known(out(i)) eq Ω)}
 + {out(i), i∈isomcomp(o)|inxa(out(i)) and argpos(i) eq 1}
 + [+: o'∈osomcomp(o)] againextractions(o')
 + [+: o'∈oanycomp(o)|if(known(o') is k) eq Ω then f
         else 1 ≤ ∃n ≤ k|o'∈ocomp(o,n)]
               o"∈againextractions(o')|known(o") eq Ω};
```

In order not to complicate the preceding equations unnecessarily, we have at a few points written less precise restrictions than could actually be applied. Most of these elisions relate to cases in which operators transform tuples of known length to tuples of unknown length.