SETL Newsletter # 187

M. Sharir
April 1977

On Inter-Procedural Flow Analysis

In this newsletter we suggest a possible modification of the data-flow maps in the presence of subprocedures, and show how these modified maps can be utilized in further global flow-analysis.

I. Introduction

The inter-procedural data-flow analysis consists of several phases. The first phase establishes the call-graph of the program, which may not be straight-forward, because of possible procedure-variables, map retrieval interpreted as a function call, partial compilation, etc. The second step obtains local information for each procedure P, including the sets defsof(P), usesin(P) and thru(P), defined in NL134 p.9. This information is already of an inter-procedural nature, and may require some iteration on the nodes of the call graph in case of cycles (recursive calls) in the graph.

The guiding principle for these phases is to obtain only safe information, i.e. - any plausible algorithm should yield only over-estimation for the edges of the call-graph, for defsof, usesin and thru.

We also assume that intra-procedural analysis has been carried out for each procedure. Taking the bfrom map, for example, we assume that for any ivariable occurence i of a variable x in any procedure P, we have a modified set, auxbfrom(i), which contains

(a) the occurences of x in P from which i can be reached via a x-free path.

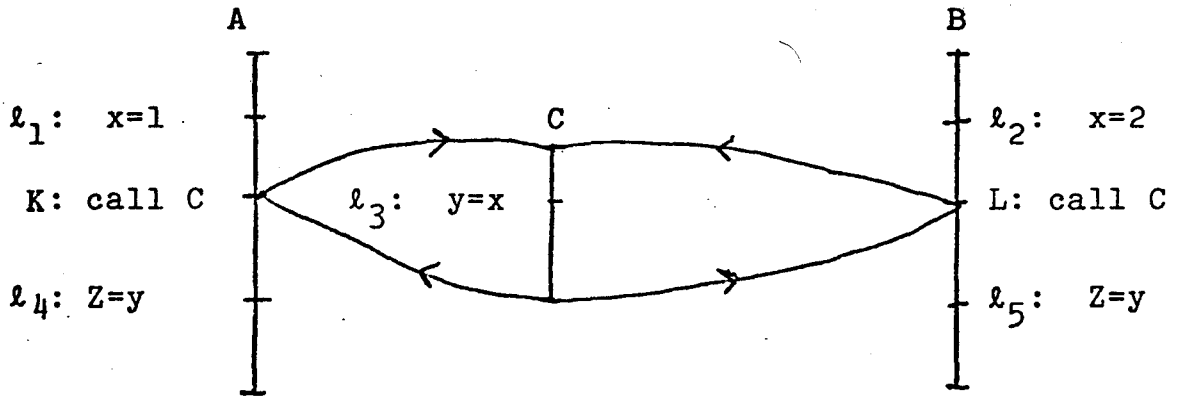
(b) the calling points in P to other procedures which might use or define x, and such that there is an x-clear path from these calling points to i.

(c) the entry point of P, if i can be reached from this point via a x-clear path.

Note that this analysis is performed simultaneously with the previous phases and may too require iteration in case of recursive procedures.

Before describing the kind of global, inter-procedural, information we'd like to derive from the above, let us first consider some examples:

(1) The simplest and most common example is - two procedures, A, B, each calling a third procedure C. The following diagram shows such a case:



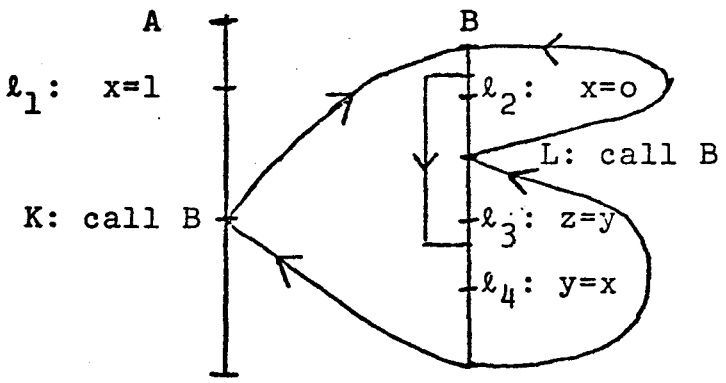
In this situation, bfrom(y_4) and bfrom(y_5) should both include y_3, and and bfrom(x_3) should include both x_1 and x_2 (We use the notation v_k to indicate the occurence of the variable v at line ℓ_k). However, during further attribute - flow analysis (type-finding, value-flow etc) we wish to avoid linking z_4 to x_2 and z_5 to x_1. (through the above chainings of the y occurences and the x-occurences).

This situation is common enough to worry about. In fact, except for stylistic reasons, the only reason to make a code fragment into a procedure is that it is invoked from several places in the program.

Remarks: (a) The same problem will occur if C is called from several points in A alone, so emphasis should be placed on particular calls, rather than more coarsely on the calling procedure.

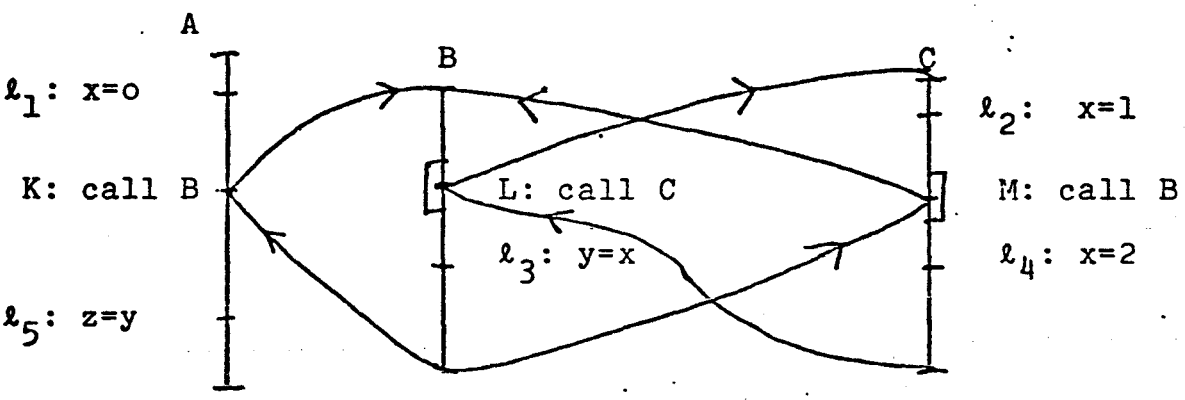
(b) If, in the above example, we replace ℓ_4 and ℓ_5 by z=x, then we could overcome our problem by replacing the bfrom map by the ud map. Indeed, realizing that x ε thru (C), we can deduce that ud(x_4) includes x_3 and x_1 but not x_2, if we only carry out an intra-procedural analysis, making use of defsof, usesin and thru of each procedure. The reason for this is that by a direct evaluation of ud we take the transitive closure of bfrom in a selective manner, making sure that after crossing to a called procedure, we go back to the same calling point only.

(2) A single recursive procedure. For example:



In this situation, bfrom(y_3) includes y_4, and bfrom (x_4) includes x_1. But y_3 or z_3 are never linked to x_1, but rather to x_2. Hence we have another instance of incorrect chaining.

(3) Co-recursive procedures. For example



Here, bfrom(y_5) includes y_3 and bfrom(x_3) includes x_1, x_2, x_4. However, z_5 can not be linked to x_2 but only to x_1, x_4.

Unlike the first example, the last two cases may be adjudged uncommon enough to ignore the problems they raise and be satisfied with the standard over-estimated data flow. Nevertheless, the algorithm that we shall now suggest will be able to handle these general cases as well.

II. The modified bfrom map and its applications.

It is clear from the examples above that some kind of trace-back information ought to be kept during attribute-flow analysis. We propose the following format for a modified bfrom map: Let v be an invariable occurence of the variable x in a procedure P. Then bfrom(v) contains all elements of the form u(w), where u is an occurence of x anywhere in the program from which v can be reached via a x-clear path, and w is a (possibly empty) string of procedure-call points, preceded by procedure-return points, such that

(a) all the procedure-calls and procedure-returns in w are executed (in the order from left to right) along the path from u to v.

(b) No other calls or returns are executed along this path, except for complete calls (i.e. - a call followed later by a return to the same place).

(c) all procedure returns in w precede all procedure-calls. We denote a call from a statement K by K itself and a return to K by K^{-1}.

Examples: Consider the above 3 examples.

(1) $\underline{\text{bfrom}}(y_4) = \{ y_3(K^{-1}) \}$

$\underline{\text{bfrom}}(y_5) = \{ y_3(L^{-1}) \}$

$\underline{\text{bfrom}}(x_3) = \{ x_1(K), x_2(L) \}$

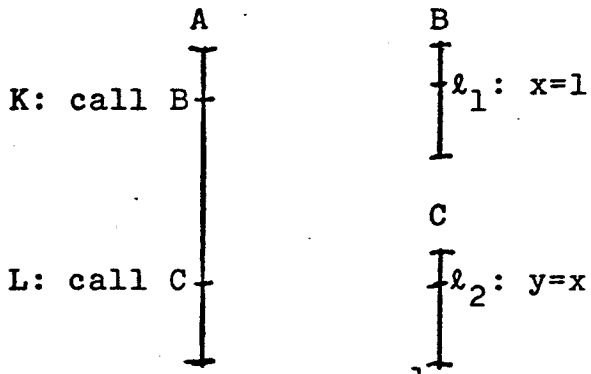
(2) $\underline{\text{bfrom}}(y_3) = \{ y_4(L^{-1}) \}$

$\underline{\text{bfrom}}(x_4) = \{ x_2(L), x_1(K) \}$

(3) $\underline{\text{bfrom}}(y_5) = \{ y_3(K^{-1}) \}$

$\underline{\text{bfrom}}(x_3) = \{ x_4(L^{-1}), x_2(M), x_1(K) \}$

(4) In the following example,



$\underline{\text{bfrom}}(x_2) = \{ x_1(K^{-1}L) \}$

Let us introduce some notations at this point. A string of calls and returns which satisfies (a)-(c) for some path in the program is called proper. i.e. - except for missing complete calls (cf. (b)), the proper strings represent all possible transfers of control between procedures along execution paths in the program. Clearly, if complete calls are ignored, then the general form of such a transfer of control is: some procedure returns followed by some procedure calls. We shall see later that the algorithm becomes more efficient if complete calls are ignored.

Additional notations are: if $w = K_1^{-1} K_2^{-1} \ldots K_m^{-1} L_1 L_2 \ldots L_n$ is a proper string, then define $\underline{\text{callpart}}\ (w) = L_1 L_2 \ldots L_n$ and $\underline{\text{returnpart}}\ (w) = K_1^{-1} K_2^{-1} \ldots K_m^{-1}$.

It is easy to obtain this bfrom map from the information already obtained (auxbfrom, defsof, usesin, thru etc.) in precisely the same way to obtain the standard bfrom. One has only to update the string of returns and calls every time procedure boundaries are crossed.

The appropriate way to utilize this modified bfrom is during any attribute-flow analysis. Suppose that the attributes appearing in such an analysis form a lattice L. Let us define a primitive trace-back attribute as a pair (α,w), where $\alpha \varepsilon L$ and w is a proper string of returns and calls. The heuristic meaning of (α,w) is as follows: the variable occurence to which (α,w) is ascribed can have attribute α if the occurence is encountered after executing a program path corresponding to w. A general trace-back attribute is defined as a finite disjunction of primitive attributes, in the form $(\alpha_1,w_1)V\ldots\ V(\alpha_n,w_n)$ with a similar meaning, and with the simplification that whenever $w_i=w_j$, we replace $(\alpha_i,w_i)V(\alpha_i,w_j)$ by $(\alpha_i V\alpha_j,w_i)$ and that if $\alpha_1=\alpha_2=\ldots=\alpha_n=\alpha$ and $w_1,\ldots,w_n$ are all the possible trace-backs for this variable occurence, we simply replace this attribute by (α,e). (Evidently there are other similar simplifications which can be applied but they would complicate the algorithm).

Let us recall that in general attribute flow algorithms propagate attributes in four basic ways (cf. V1). We shall show how to modify each propagation rule when the modified bfrom map that we have described is available:

I. FWD(←): In this case we have an instruction $o = i_1$ op i_2, and we want to find the attribute of o from the attributes of i_1 and i_2. Suppose that (α_1,w_1), (α_2,w_2) are primitive attributes of i_1,i_2 respectively, and that α = forward $(\alpha_1,\alpha_2,$ op) is the standard attribute derived for o in this instruction.

Let us say that w_1 and w_2 are compatible if there exist two corresponding paths in the program flow such that one of them is a terminal subpath of the other. Thus, in example 3, KLM and M are compatible, whereas KLM and K are not.

Thus, if w_1 and w_2 are compatible, and w_2 has a shorter program path than w_1, then as we execute the path corresponding to w_1, both attributes α_1, α_2 become effective, and hence (α,w_1) can be correctly taken as a primitive attribute of o. It is also clear that if w_1 and w_2 are not compatible, then no inference about the attribute of o can be made from α_1 and α_2. Let us denote by $\max(w_1,w_2)$ the one with the larger program path, and similarly define $\min(w_1,w_2)$.

In this way the attribute of o is computed. If (α_1,w_1) is a primitive attribute of i_1 for which there is no primitive attribute (α_2,w_2) of i_2 such that w_1,w_2 are compatible, we add $(0,w_1)$ as a primitive attribute of o (0 is the zero, or error attribute).

Note that in the second pass of the typefinder, e.g., we will want to compute the conjunction of the attribute of o with the above-calculated attribute. The appropriate rules for this are as follows:

(a) $$[(\alpha_1,w_1)\vee\ldots\vee(\alpha_n,w_n)]\wedge[(\beta_1,z_1)\vee\ldots\vee(\beta_m,z_m)]$$

$$=\bigvee_{i,j}[(\alpha_i,w_i)\wedge(\beta_j,z_j)]$$

(b) If w_i and z_j are compatible, and, say, w_i is their minimum and z_j is their maximum, then

$$(\alpha_i,w_i)\wedge(\beta_j,z_j)=(\alpha_i\wedge\beta_j,z_j)\vee(\alpha_i,w_i)$$

otherwise, the conjunction $(\alpha_i,w_i)\wedge(\beta_j,z_j)$ is ignored.

Exactly the same rules apply to the BACK($\rightarrow$) propagation, where the attribute of an ivariable is determined by the attributes of the other occurences in the instruction.

Before proceeding further, let us make some comments on compatibility. Let w, z be two proper strings, and let α=callpart(w), β=callpart(z). The somewhat weaker condition that we shall use for compatibility is:

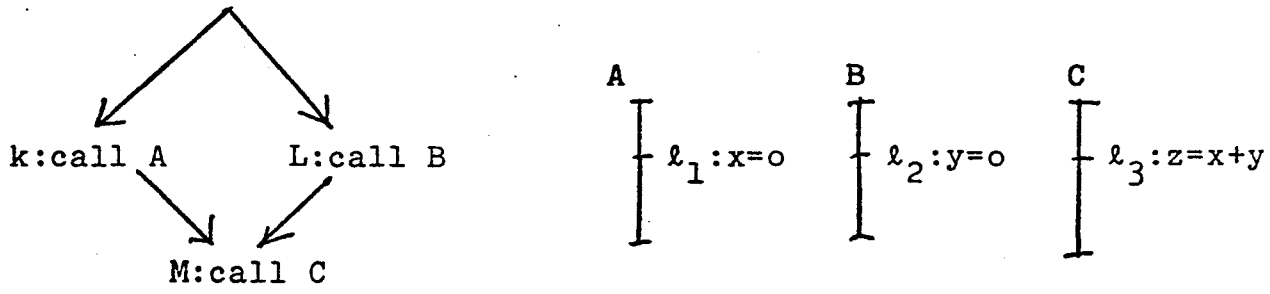
(a) If $\alpha=\beta$ then w and z are compatible. Any of them may be taken as their minimum or maximum.

(b) If $\alpha\neq\beta$ and α is a terminal substring of β, then w and z are compatible, min(w,z)=w and max(w,z)=z.

(c) Otherwise w and z are not compatible.

This criterion is necessary but by no means sufficient, and may lead to an over-estimation, as the following example indicates:

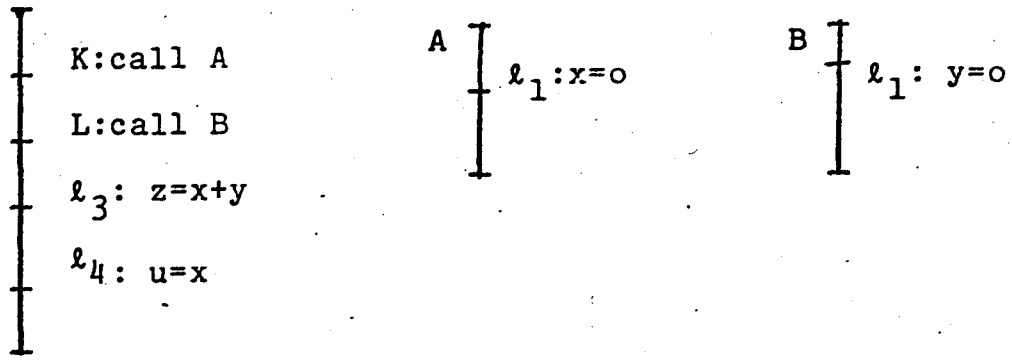
(5)



Here bfrom(x_3) = {$x_1(K^{-1}M)$} and bfrom(y_3)= {$y_2(L^{-1}M)$}. According to the above criterion, $K^{-1}M$ and $L^{-1}M$ are compatible, though there is actually no program path for which z is defined.

The trouble stems from the fact that we have ignored complete calls in the proper strings. If we allow trace-back strings which contain also complete calls, i.e. strings like $K^{-1}LL^{-1}M$, then the correct criterion for compatibility is that one string is a terminal substring of the other. In this case, $K^{-1}M$ and $L^{-1}M$ in example 5 are not compatible However, in the following example:

(6)



we would have, according to the tentative suggestion just made,

$$\underline{\text{bfrom}}\ (x_3) = \{x_1(K^{-1}LL^{-1})\}$$
$$\underline{\text{bfrom}}\ (y_3) = \{y_2(L^{-1})\}$$

and we shall correctly determine that $K^{-1}LL^{-1}$ and L^{-1} are compatible. However, including strings with complete calls would complicate the algorithm considerably, since we shall have to keep and calculate redundant trace-back information for many variables, for the chance that they might interact with variables for which this trace-back is meaningful. Indeed, in example 6, u_4 will have the attribute of x_1 with the trace-back $K^{-1}LL^{-1}$ which is clearly not useful. Besides, our tentative suggestion complicates the computation of bfrom, since we would also have to consider branching to procedures which do not modify the variable at all!

For these reasons, we propose to accept the over-estimation produced by the above criterion for compatibility. There is another way, however, to overcome the above problem. This is to compute for each pair of calling points in the same procedure, which of them can be the successor of the other in the flow-graph. Then the correct criterion for compatibility is as follows:

(a) If $\alpha=\beta$, let returnpart$(w) = J_m^{-1}J_{m-1}^{-1}\cdots J_1^{-1}$ and returnpart $(z)=K_n^{-1}K_{n-1}^{-1}\cdots K_1^{-1}$. Let r be an index such that $K_i=J_i$ for all $i<r$, and $K_r\neq J_r$. If no such r exists, then it is obvious that one of w,z is a terminal substring of the other and they are compatible, with obvious definitions of their maximum and minimum. If such r exists, then it is easy to check that K_r and J_r occur within the same procedure. Now, if one of them can be the successor of the other, then w and z are compatible, and if say K_r follows J_r in the flow-graph, then max(w,z)=w and min(w,z)=z. On the other hand, if neither of K_r,J_r can follow the other, then w and z are not compatible. (b) and (c) are the same as above.

II. BFRM (↓): In this case we calculate the attribute of an ivariable v as the disjunction of the attributes of all occurences in bfrom(v). The specific rule applied is as follows:

Suppose that (α,w) is a primitive attribute of $u(z) \in$ bfrom(v). We say the z is a proper continuation of w if, after concatenating w||z, and deleting from it iteratieely all consecutive pairs of the same call and return (i.e. of the form KK^{-1}) we are left with a proper string of returns and calls. This string is denoted by $w\cdot z$. Then $(\alpha,w\cdot z)$ is taken as a primitive attribute of v. Otherwise (α,w) is ignored.

Example (1): Let attrib(x_1)=(1,e) , attrib(x_2)=(2,e) (1,2 are any attributes and è is the empty string)

Then
attrib(x_3) = (1,K) V (2,L) (BFRM)
attrib(y_3) = (1,K) V (2,L) (FWD)
attrib(y_4) = (1,e) (BFRM)
attrib(y_5) = (2,e) (BFRM)

Thus, the incorrect chainings have been avoided.

(2) Let $attrib(x_1) = (1,e)$

$attrib(x_2) = (0,e)$

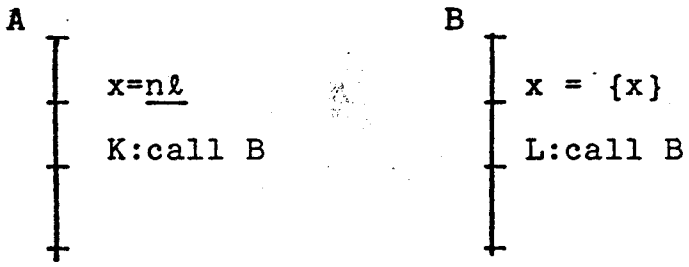
$attrib(x_4) = (0,L) \vee (1,K)$ (BFRM)

$attrib(y_4) = (0,L) \vee (1,K)$ (FWD)

$attrib(y_3) = (0,e)$ (BFRM)

and again the correct chaining has been obtained. It can be easily verified that in example 3 too the algorithm yields the correct chaining.

Note that if no bounds are imposed in flow-analysis we can obtain arbitrarily long strings. e.g. -



If in this example we do not restrict the nesting level of sets allowed, e.g., in type-finding we can produce attributes for x in B with strings of arbitrarily many L's. However, for practical purpose, it is reasonable to suppress cyclic strings in the trace-back, i.e. - regard KLL as KL etc.

III. FFRM (↑): In this case we aim to obtain the attribute of an occurence v as the disjunction of the attributes of all occurences u such that $v \in \underline{bfrom}(u)$, i.e. - $u \in \underline{ffrom}(v)$, where the actual modified definition of ffrom is:

$$\underline{ffrom}\ (v) = \{\ u(z) :\ v(z) \in \underline{bfrom}(u)\ \}$$

Here we apply the follwing rule:

Suppose that (α,w) is a primitive attribute of $u(z)\varepsilon$ ffrom(v). Then, if $z=K_1^{-1}K_2^{-1}\ldots K_m^{-1}L_1\ldots L_n$, put $z^{-1}=L_n^{-1}K_m\ldots K_1$. If z^{-1} is a proper continuation of w, then $(\alpha,w.z^{-1})$ is taken as a primitive attribute of v. Otherwise (α,w) is ignored.

Again, this sometimes leads to an over-estimation. In example (5), $(0,K^{-1}M)$ is a primitive attribute of x_3 and $x_3(L^{-1}M)\ \varepsilon$ ffrom(x_2). We shall thus conclude that $(0,K^{-1}L)$ is a primitive attribute of x_2, which is incorrect. Note, however, that if $K^{-1}L$ were proper, then we would obtain a correct attribute.

Remarks:

(a) The algorithm that has been suggested might also be useful in intra-procedural analysis, in particular, it can detect errors of a sort that would otherwise go undetected when at an instruction we unknowingly merge two flow paths which are not simultaneously executable. We can thus generalize the algorithm and carry trace-back information during regular flow analysis to obtain deeper results whenever it is desirable to do so.

(b) If, after completing the analysis, we find variable occurences with two or more possible attributes, we might produce conditional code which, depending upon the calling instruction from which the procedure was last called, utilizes the corresponding attribute of the occurence efficiently.

(c) Actually, a simpler, slightly over-estimating algorithm could be obtained if we define the trace-back strings to include procedure calls only (without returns), while retaining the same form of the bfrom map. This would make it easier to handle the strings, check for compatibility etc. while the loss of information will be about the same as in the above algorithm. More specifically - two strings w,z will be compatible if one is a terminal substring of the other, with natural simplified definitions of their maximum and minimum, whereas the appropriate rules for the BFRM and FFRM propagations remain the same.

References

[NL134] J. Schwartz, Inter-Procedural Optimisation, SETL Newsletter 134.

[V1] L. Vanek, Global Analysis Techniques for the optimizing SETL Compiler.