APPLICATION OF LATTICE ALGEBRA TO LOOP OPTIMIZATION+

Amelia Fong, John Kam and Jeffrey Ullman
Department of Electrical Engineering
Princeton University
Princeton, N.J. 08540

I. Introduction

Kildall [1] has recently developed lattice theoretic techniques for solving many data flow analysis problems. It is the purpose of this paper to demonstrate that many of the loop optimizations such as 'code motion' and induction variables detection can be done efficiently and in great generality by essentially the same lattice theoretic techniques.

We shall use the usual model for a program being subjected to code improvement, the flow graph $G = (N, E, n_0)$, where N is a set of nodes, E is a set of edges and $n_0 \in N$ is the initial node. There is a path from n_0 to each node in N. The nodes represent straight line blocks of code. There is an edge from n_1 to n_2 if n_2 can immediately follow n_1 in a possible execution of the program.

The notion of a loop in a flow graph may be modeled by a region $R = (N_1, E_1, n_1)$ which is a set of nodes N_1, and edges E_1 with a header node n_1 having the property that every path from the initial node to a node in R passes through n_1. Various loop optimization techniques such as "code motion" or induction variables (see [2,3,7], e.g.) make use of functions relating information at the entry of a region to the information at the nodes of the region. This idea can be generalized to lattices as developed by Kildall.

We shall use the following formulation of Kildall's ideas.

Definition: A data flow analysis framework is a pair $D = (L, F)$ where

(i) L is a semilattice with meet $\wedge$ and zero element $\underline{0}$, satisfying the boundedness condition

$$(\forall x \in L)(\exists k)(x_1 < x_2 < \ldots < x_n = x \text{ implies } n \leq k)$$

where $y \leq z$ is shorthand for $y \wedge z = y$ and $y < z$ means $y \leq z$ and $y \neq z$.

(ii) F is a set of functions from L to L (morphisms) on L) closed under composition and meet, having an identity (denoted by e) and satisfying the distributivity condition

$$(\forall f \in F)(\forall x, y \in L)(f(x \wedge y) = f(x) \wedge f(y)).$$

(iii) For each $x \in L$, there exists $f \in F$ such that $x = f(\underline{0})$.

Intuitively, the lattice elements represent information which might be known about data at entrance to some block of a flow graph, and F represents the set of transformations on this information that could be effected by portions of a program (basic blocks in particular) as control passes through it. In the case of loop optimization, we shall be interested in functions relating information at the entry of a region to that at each node of the

+ Work supported by NSF grant GJ-1052.

region. These functions may be defined as follows. For each region R and each node n in R, define the function $f_{R,n}$ such that for all x in L

$$f_{R,n}(x) = \bigwedge_{P} f_P(x)$$

where the meet is taken over all paths P in R from the header of R to the exit of n. The <u>function f_P associated with a path P</u> is the composition of the functions which reflect the actions of the nodes on the path.

In certain cases an efficient algorithm can be obtained to compute the functions associated with various regions as a flowgraph is parsed [4] or interval analyzed [5]. The issues which we must consider to demonstrate the practicality of our approach are:

1) How can morphisms be represented so that their important operations - composition and meet ($f \wedge g$ is defined by $[f \wedge g](x) = f(x) \wedge g(x)$) can be performed efficiently?

2) Under what conditions are the functions associated with regions efficiently computable?

3) Are there frameworks that meet the requirements of (1) and (2) and have practical applications?

We shall turn to each of these issues in the following sections.

In Section II, we shall discuss examples of representations of functions. In Section III, we shall give the outline of an efficient algorithm to compute these functions and discuss the conditions under which the techniques in the algorithm may be applied. Section IV concludes the paper by giving some practical applications of this approach.

II. <u>Representing Functions</u>

For the kinds of lattices L and sets of morphisms F on L considered most frequently, representation of the functions appears to be feasible. In fact the representation of the morphisms in each case bears a resemblance to the representation of a pair of elements of L, but we cannot prove a general result of this nature.

The most common case considered in the literature on global flow analysis [2,3,4,6] is the use of bit vectors. Here L is the set of bit vectors of some fixed length, $\wedge$ is bitwise "and," and the elements of F are functions which deal with the components of bit vectors uniformly and independently. In this case each f in F can be represented as a pair of bit vectors a and b, where $f(x) = (a \wedge x) \vee (b \wedge \neg x)$. Examples where this approach has been used are the GEN and KILL representation of [3,4], or the "dual assumptions" approach of [2].

For a second example, in [1], the "structured partition" lattice is described for common subexpression detection. A relatively efficient representation of structured partitions in terms of "value numbers" [7] was given in [1]. The representation is equivalent to the "dag" introduced in [8]. We shall show that the dag provides a natural generalization to representation of morphisms on structured partitions.

<u>Definition</u>:

Let A be a finite set of variables $A = \{X_1,\ldots,X_k\}$.

Let $\bar{A} = \{\bar{X}_i \mid X_i \in a\}$.

Let C be an infinite set of constants.

Let O be a finite set of binary operators.

A <u>structured partition DAG</u> $G = (V,E,LABEL_G,OP_G)$ with respect to (A,C,O) is a finite directed acyclic graph with a set of nodes V, a set of edges E, together with two functions

$LABEL_G$: $V \rightarrow$ set of finite subsets of $(A \cup \bar{A} \cup C)$

OP_G : $V-B \rightarrow O$ where B is the set of base nodes

satisfying the following conditions

1) No symbol or constant is in the two sets labelling two nodes of the dag.

2) Each $\bar{X}_i \in \bar{A}$ appears in the label of exactly one base node.

3) If v is a base node, $LABEL_G(v)$ contains either one $\bar{X}_i \in \bar{A}$ or one constant $c \in C$ but not both. (It may contain any number of symbols in A)

Intuitively, if the dag represents a function $f_{R,n}$, barred symbols represent initial values on entry to region R and unbarred symbols represent final values on exiting node n. For example the function associated with the block:

$$A \leftarrow B + C$$
$$B \leftarrow A * D$$
$$A \leftarrow C + 1$$

would be represented by the DAG of Fig. 1.

Fig. 1

It is clear from the above example that structured partition dags (SPDAG) representing functions associated with straight line blocks can be obtained in a routine fashion. Composition of functions can also be performed on these dags in the following manner.

Let f_1 and f_2 be the functions associated with the paths p_1 and p_2 respectively. Let G_1 and G_2 be the corresponding SPDAG's. $f_2 \circ f_1$ is the function associated with the path consisting of p_1 followed by p_2. Hence final values of G_1 should become initial values of G_2 in the composite SPDAG G_3 representing $f_2 \circ f_1$. To create G_3 from G_1 and G_2, nodes of G_1 are linked with base nodes of G_2 by identifying each unbarred symbol X of G_1 with the corresponding barred symbol $\bar{X}$ of G_2. The two nodes are collapsed into one node and the symbols X and $\bar{X}$ are deleted from the label of the resulting node, since the X of G_2 is now the final value and the $\bar{X}$ of G_1 is now the initial value for $f_2 \circ f_1$. An example will illustrate the operation. G_1 and G_2, together with the required node mergers are shown in Fig. 2(a), while Fig. 2(b) shows the resulting DAG G_3.

(a)

(b)

Fig. 2

A binary operation on the SPDAG's must be defined to reflect the meet operation on functions.

<u>Definition</u>:

Let $G = (V,E,LABEL_G,OP_G)$ be an SPDAG with respect to (A,C,O). $G' = (V',E',LABEL_{G'},OP_{G'})$ is a <u>sub-SPDAG</u> of G if

$V' \subseteq V$ and V' includes all base nodes of G,

$E' = E \cap V' \times V'$

and for all $v' \in V'$

$LABEL_{G'}(v') \subseteq LABEL_G(v')$

$OP_{G'}(v') = OP_G(v')$

For example, in Fig. 3, G' is a sub-SPDAG of G.

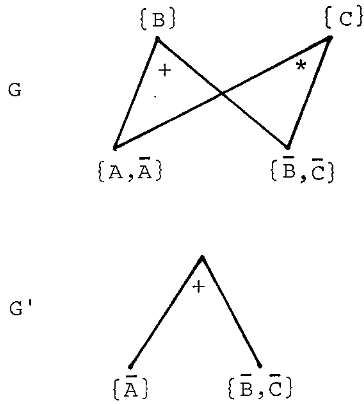

Fig. 3

Definition

Let G_1, G_2 be two SPDAG's with respect to (A,C,O). G_1 is <u>isomorphic</u> to G_2 if there is a 1-1 correspondence 2 between the nodes V of G_1 and the nodes W of G_2 such that for all $v \in V$, if $g(v) = w \in W$ then

1) v is a base node iff w is a base node and $LABEL_{G_1}(v) = LABEL_{G_2}(w)$, and

2) if v, w are not base nodes, then $OP_{G_1}(v) = OP_{G_2}(w)$ and the left and right sons of v are in correspondence with the left and right sons of w.

Let G_1 and G_2 be two SPDAG's representing f_1 and f_2, then $f_1 \wedge f_2$ is represented by the maximal SPDAG G_3 which is isomorphic to a sub-SPDAG of G_1 and a sub-SPDAG of G_2. The definition of isomorphism can easily be executed to take into accout the commutativity of certain operators, but we do not do so here.

It should be clear that the meet of SPDAG's is effectively computable. The following algorithm constructs the meet G_3 of two SPDAG's G_1 and G_2 w.r.t. (A,C,O) in an efficient manner.

First, we need the definition of the rank of a SPDAG.

<u>Definition</u>: The <u>rank</u> of a base node is 0.

The <u>rank</u> of an interior node is the maximum rank among its sons plus 1.

The rank of an SPDAG G denoted RANK(G), is the maximum rank of its nodes.

Algorithm

Let V and W be the set of nodes of SPDAG's G_1 and G_2 whose meet we want to compute.

<u>Initialization</u> - <u>consider</u> <u>base</u> <u>nodes</u>.

Assign integers to operators in O such that for each operator $o \in O$, there is a unique integer associated with it.

Let $A = \{X_1, X_2, \ldots, X_k\}$

There are k base nodes on G_1 and on G_2 each has label containing one $\bar{X}_i$ while the remaining base nodes have labels containing constants. Let $C' \subset C$ be the finite set of constants which appear in the labels of G_1 and G_2.

Define a linear order <· on $\bar{A} \cup C'$ such that $\bar{X}_i <\cdot \bar{X}_j$ for all $i < j$, $1 \leq i, j \leq k$

$\bar{X}_i <\cdot c$ for all i, $1 \leq i \leq k$, for all $c \in C'$

$C_1 <\cdot C_2$ if C_1 is less than+ C_2, for all $C_1, C_2 \in C'$

Represent each base node in G_1 by the barred variable or constant contained in its label. Sort the base nodes on this representation according to the linear order <· . Sort the base nodes of G_2 in the same manner. Merge the two sorted lists, obtaining the pairs of base nodes (v_i, w_i), $v_i \in V$, $w_i \in W$ which have the same representation.

Let the total number of these pairs be m_0.

Construct m_0 base nodes $\{u_1, u_2, \ldots, u_{m_0}\}$ for G_3 such that if $(v_1,w_1), (v_2,w_2) \ldots, (v_{m_0}, w_{m_0})$ are the pairs obtained from the merge, then for all $1 \leq i \leq m_0$

$$LABEL_{G_3}(u_i) = LABEL_{G_1}(v_i) \cap LABEL_{G_2}(w_i).$$

+ We assume C has an ordering defined on it. This will surely be the case for any constants that are computer representable.

If $V_0 = \{v_1, v_2, \ldots, v_{m_0}\}$ $\{w_1, w_2, \ldots, w_{m_0}\}$ and $\{u_1, u_2, \ldots, u_{m_0}\}$ are the sorted lists of nodes described above, define $NUM_{G_1}(v_i) = i$, $NUM_{G_2}(w_i) = i$ and $NUM_{G_3}(u_i) = i$.

Induction Step

We now present the algorithm which considers nodes in order of increasing rank, constructing the SPDAG G_3 as it goes.

1) Initialize

$total \leftarrow m_0$;
$r \leftarrow 0$;

2) Increment rank

$r \leftarrow r+1$

3) Consider nodes of rank r.

Let V_r be the set of nodes of G_1 of rank r. Define W_r similarly.

if either $V_r = \emptyset$ or $W_r = \emptyset$ then stop

4) Assume $V_r \neq \emptyset$ and $W_r \neq \emptyset$. For each node v in V_r, form the 3 component key (k, n_1, n_2) where k, n_1, n_2 are all integers. k is the integer associated with $OP_{G_1}(v)$, $n_1 = NUM_{G_1}(v_1)$, $n_2 = NUM_{G_1}(v_2)$ where v_1, v_2 are the left and right sons of v respectively.

Similarly, for each node w in W_r, form the three component key in the same fashion. Sort V_r and W_r separately on the three component key. Merge the two sorted lists, obtaining the pairs of nodes (v_i, w_i) $v_i \in V_r$ $w_i \in W_r$ which have exactly the same key. Let the number of these pairs be m_r.

if $m_r = 0$ then stop

5) Assume $m_r > 0$. Let (v_i, w_i) $1 \leq i \leq m_r$ be the pairs obtained from the merge. Create nodes $U_r = \{u_1, u_2, \ldots, u_{n_r}\}$ on G_3 such that the left and right sons of each u_i are the nodes on G_3 corresponding to the left and right sons of v_i (also w_i) i.e. the number assigned to the corresponding nodes are the same.

For all $1 \leq i \leq m_r$

$OP_{G_3}(u_i) = OP_{G_1}(v_i) = OP_{G_2}(w_i)$

$LABEL_{G_3}(u_i) = LABEL_{G_1}(v_i) \cap LABEL_{G_2}(w_i)$

Let $V_r = \{v_1, v_2, \ldots, v_{m_r}\}$, $W_r = \{w_1, w_2, \ldots, w_{m_r}\}$ and $U_r = \{u_1, u_2, \ldots, u_{m_r}\}$. Number the nodes in V_r of G_1 such that $NUM_{G_1}(v_i) \leftarrow total + i$, $1 \leq i \leq m_r$. Similarly number W_r and U_r such that $NUM_{G_2}(w_i) \leftarrow total + i$, $NUM_{G_3}(u_i) \leftarrow total + i$, $1 \leq i \leq m$.

$total \leftarrow total + m_r$

go to step (2) .

By using a bucket sort [12], the meet of two SPDAG's may be obtained in time proportional to the total number of nodes in the two SPDAG's. The algorithm can be easily modified to accommodate a definition of isomorphism taking into account the commutative laws of certain operators.

III. Computing Functions Efficiently

In this section we shall outline an efficient algorithm adapted from [4], to compute functions for progressively larger regions in terms of the functions for the constituent regions, and discuss the conditions under which the techniques in the algorithm is applicable.

The idea is to construct a rooted, unordered tree representing each region, each leaf representing a node in the region and with the following properties:

(i) Each interior node has two or three sons, except that a two node tree is permissible.

(ii) All paths from a node to its descendant leaves have the same length.

The edges of the tree will be labelled by a function. It will be arranged so that if the tree represents region R, then $f_{R,n}$ can be computed by following the path from the leaf n to the root and composing the functions labelling the edges of that path, bottommost function first.

A basic manipulation of edge labels is given in the next lemma and exhibited in Fig. 4. We refer to it as stripping of an edge (t_0, t'_0).

Let t_0, t'_0 and $t_1, \ldots, t_k$ be nodes of a tree with edges and labels as in Fig. 4(a). If the labels of these edges are changed to those in Fig. 4(b) then the same $f_{R,n}$ as before is computed for all the leaves of the tree.

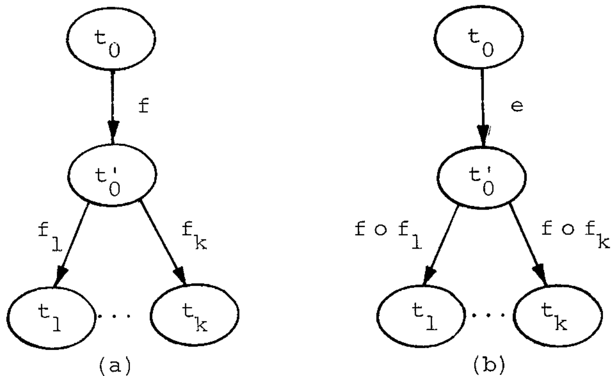

Fig. 4

Note that an edge with label e, the identity function, has no effect on the calculation of $f_{R,n}$ when the path from the loop representing node n to the root is followed.

The algorithm consists of three parts, one for initialization, i.e. the construction of trees for regions consisting of a single basic block, the second for regions constructed by applications of T_1 and T_2 where T_1, T_2 are transformations on flow graphs defined in [13]. T_1 is the deletion of an edge from a node to itself. T_2 is the merging of n_1 and n_2 into a single node, where n_1 is the unique predecessor of n_2 and n_2 is not the initial node. We say n_2 is consumed by n_1.

We assume that the flow graph is reducible (i.e. it can become a single node under the T_1, T_2 transformations [13]) and that a sequence of reductions by T_1 and T_2 are available. It should be emphasized that this algorithm will not work for an arbitrary framework, but only on a restricted class to be defined subsequently.

Algorithm

1) Initialization

For the initial regions consisting of a single node n, construct the tree as shown in Fig. 5.

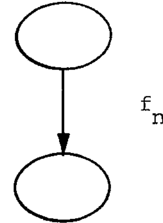

Fig. 5

2) Application of T_1

Suppose region $R_1 = (N, E_1, n_1)$ is created by an application of T_1 from region $R_2 = (N, E_2, n_1)$. That is, E_1 consists of E_2 and those edges of the original flow graph represented by the edge eliminated.

$$\text{Let } f = \bigwedge_{(\ell, n_1) \in E_1 - E_2} f_{R_2, \ell}$$

(i times). Create the tree for region R_1 by the following steps:

1) Create a new node r whose lone son is the root r' of the tree for R_2. Label edge (r,r') by f.

2) Strip the edge (r,r').

3) Delete r and the edge (r,r'); r' is the root for R_1.

3) Application of T_2

Let $R = (N, E, n_1)$ be the region formed by T_2 from regions $R_1 = (N_1, E_1, n_1)$ and $R_2 = (N_2, E_2, n_2)$, with R_1 consuming R_2.

$$\text{Let } f = \bigwedge_{(\ell, n_2) \in E - E_1 - E_2} f_{R_1, \ell}$$

Create the tree for region R by the following steps:

1) Create a new node r whose lone son is the root r' of the tree for R_2. Label the edge (r,r') by f.

2) Strip the edge (r,r')

3) Delete r and the edge (r,r')

4) Merge the resulting tree with root r' and the tree for region R_1. The resulting tree is the tree for region R.

The merger algorithm is rather complicated, but is identical in spirit to that of [4]. We omit the details.

The necessary and sufficient condition for the above algorithm to work is for the meet of the functions associated with all paths in R' from its header to a node n in R to be equal to the meet of the functions associated with a particular restricted set of paths. This restricted set consists of those paths which include at most one back latch. This condition is equivalent to:

1) $(\forall f,g,h \in F)(hfg \geq hf \wedge hg \wedge h)$

In turn, we can easily show the equivalence of (1) to the simpler condition

2) $(\forall f \in F)(ff \geq f \wedge e)$

Condition (2), in its turn is implied by:

3) $(\forall f,g \in F)(\forall x \in L)(fg(\underline{0}) \geq f(x) \wedge g(\underline{0}) \wedge x)$

which is the condition shown in [9] to be necessary and sufficient for the depth first search technique of [10] to be applicable to a data flow analysis framework. (2) does not imply (3), however [9].

The reader may easily check that the structured partition dag of Section II does satisfy condition (2).

We can also show that the generalization of (3):

4) $(\exists k)(\forall f \in F)(f^k \geq \bigwedge_{i=0}^{k-1} f^i)$

is sufficient for a straightforward generalization of the algorithm described above to work. This algorithm requires $O(n \log n)$ composition and meet operations on an n node flow graph, as does the algorithm above.

It should be noted that Kildall's conditions are not sufficient by themselves to guarantee that any algorithm for computing morphisms will converge. The problem is that boundedness must be replaced by a "uniform boundedness" condition, else there would be no limit on the number of passes around a loop needed for a function to attain its final value for every value of its argument.

IV. Some Applications

A. Loop invariant computations. An assignment $A \leftarrow B \theta C$ is loop invariant if neither B nor C change inside the loop. In terms of regions, each path inside the region from the header to the assignment must leave B and C unchanged. If the region happens to have some back latches, then it is efficient to move the computation $A \leftarrow B \theta C$ to just before the header.

One easy way to detect many loop invariant computations is to use a lattice of bit vectors with one position for each variable. The bit for a variable is to be 1 at a point if and only if no path in the region from the header to the point sets the variable. Meet is logical "and," and the function associated with $A \leftarrow B \theta C$ sets the bit for A to 0 and leaves others unchanged.

A more sophisticated approach is to use the structured partition representation described in Section II. This approach enables us to compute $f_{R,n}(\underline{0})$ for each region R and node n in R. If $f_{R,n}(\underline{0})$ is a dag including a structure such as the one in Fig. 6 we could detect that B is

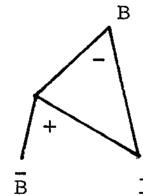

Fig. 6

invariant at the entry to n, although the bit vector approach would not do so.

B. Induction variables. Intuitively, an induction variable X at a node n in region R is one for which every path in R from the header to the exit of n adds the same loop invariant quantity (increment) to X. Alternatively, successive values of X at n form an arithmetic progression as long as we stay within the region R. If for all nodes n in R which are predecessors of the header (tails of back latches) X is an induction

variable at n with the same increment, then X is an <u>induction variable of R</u>. It is these variables we wish to identify, but the more general search for induction variables at individual nodes n seems to be the most efficient route to our goal.

We can again use the structured partition lattice to compute a dag representing $f_{R,n}(\underline{0})$ for each R and n. We may then identify induction variables at n by a variety of rules, depending on how much algebraic manipulation we are willing to perform. The basic strategy is to look for variables X such that $f_{R,n}(\underline{0})$ indicates that X is incremented by a constant each time through n. An example is Fig. 7(a), where $X = \bar{X} + 2$. For a more subtle example, consider Fig. 7(b). Suppose Y is an induction variable of the region R. (We can only tell this by considering Y at all the latching nodes of R.) Then, since X has a value which is a linear function of an induction variable, namely 2Y-1, it can be shown to assume an arithmetic progression at n.

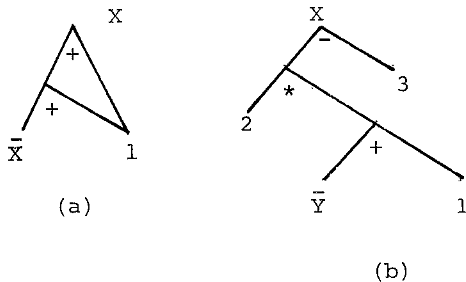

Fig. 7

C <u>Type Discovery - Self Dependency</u>. Another interesting application of these techniques is to detect the self dependency of a variable. We shall discuss one situation where this idea is useful. Tennenbaum [11] has used lattice techniques to discover identifier types in SETL [11]. Here the lattice elements are sets of types which variables could assume. Meet is union of sets of types, and the functions associated with blocks reflect certain inferences regarding types which may be made from the syntactic rules of SETL. Unfortunately, the lattice of all sets of types is not bounded. For example, the piece of flowchart in Fig. 8 gives X the infinite set of types "integer or set of integers or set of set of set of integers, or ... "

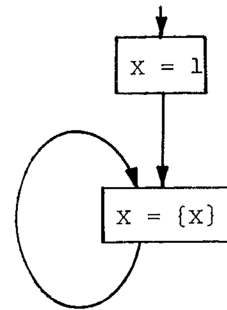

Fig. 8

The approach taken in [11] is to limit the depth of nesting of "set of" or "tuple of" to three. Anything more complex is "don't know," i.e., the lattice $\underline{0}$. We can avoid any a priori bound on the depth of nesting of type descriptors if we detect those variables whose type depends nontrivially on itself around a loop (by nontrivially, we mean that a set or tuple forming operation is involved in the formula that relates the type after traversing the loop to the type before the loop). If we detect only these variables - and we can do so using a variety of techniques - then we can give each of these variables the type $\underline{0}$ around the loop before applying global propagation techniques as [11] does.

In this way, the lattice of types will still not obey Kildall's boundedness condition, but there will be a bound for every flow graph. Thus Kildall's technique can be made to work even though his condition is, strictly speaking, not satisfied.

We can represent functions on types by a notation similar to the dag representation discussed in Section II. The symbols $\bar{X}$ and X represent the set of possible types for X at entrance to the region and "currently." The constants are a-priori defined types such as integer or character string. The operators, if it is SETL we are talking about, are { } (set former), < > (tuple former), and | (alternation or union of types). For example the dag in Fig. 9 says that the type of X "currently" is either

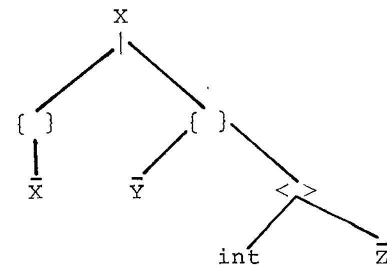

Fig. 9

1) a set of whatever types of elements $\bar{X}$ could represent at entrance to the region, or

2) a set of elements which are each either of whatever type Y could be initially or 2-tuples consisting of an integer and an object of whatever type $\bar{Z}$ was initially.

The meet operation in this lattice is alternation; the reader can easily construct an algorithm to perform the meet operation on dags by splicing them together with new nodes labeled | . The effect of basic blocks on types of variables is as described in [11]. It may be easily checked that condition (4) of Section III is satisfied if k is the number of variables.

In order to determine all and only those variables which could assume an infinite set of types when the program was run, we would have to concern ourselves with actual values assumed by variables, rather than their types alone. However, we can, using the dag representation defined above, find a superset of all such varaibles. Namely, if R is a region with header n_0, then say X is self dependent if in the dag representing f_{R,n_0}, there is a path from $\bar{X}$ to X which contains a node labeled by the { } or < > operator.

References

[1] G.A. Kildall, "A Unified Approach to Global Program Optimization," Proc. ACM Symposium on Principles of Programming Languages, pp 194-206, October 1973.

[2] M. Schaefer, A Mathematical Theory of Global Program Optimization, Prentice Hall, 1973.

[3] A.V. Aho and J.D. Ullman, The Theory of Parsing, Translation and Compiling Vol. II, Prentice Hall, 1973.

[4] J.D. Ullman, "Fast Algorithms for the Elimination of Common Subexpressions," Acta Inf., 2, pp 191-213, January 1974.

[5] F.E. Allen, "Control Flow Analysis," SIGPLAN Notices, 5:7, pp 1-19, July 1970.

[6] J. Cocke, "Global Common Subexpression Elimination," ibid. pp 20-24.

[7] J. Cocke and J. Schwartz, Programming Languages and Their Compilers, Courant Inst., N.Y.U., New York, 1970.

[8] A.V. Aho and J.D. Ullman, "Optimization of Straight Line Programs," SIAM J. Computing, 1:1, pp 1-19, March 1972.

[9] J. Kam and J.D. Ullman, "Global Optimization Problems and Iterative Algorithms," TR 146, Computer Science Lab., Dept. of Electrical Eng'g, Princeton University, N.J., January 1974.

[10] M.S. Hecht and J.D. Ullman, "Analysis of a Simple Algorithm for Global Flow Problems," Proc. ACM Symposium on Principles of Programming Languages, pp 207-217, October 1973.

[11] A. Tennenbaum, Type Determination in Very High Level Languages," NSO-3, Courant Inst., N.Y.U., New York, 1974.

[12] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, Reading, Mass., 1974.

[13] M.S. Hecht and J.D. Ullman, "Flow Graph Reducibility," SIAM J. Computing 1:2, pp 188-202, June 1972.

[14] J.T. Schwartz, On Programming Vols I and II, Courant Institute, New York, 1973.