A DATA-FLOW ANALYSIS METHOD

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements of the Degree

Master of Science

---

by

Kwang Cook Lim

December 1976

ACKNOWLEDGMENT

I would like to express my gratitude to Dr. Jung-Chang Huang for his invaluable guidance throughout the development of this material. My appreciation is also extended to Dr. Olin Johnson and Dr. Betty Barr for their interest and participation in this project. Finally, a special thanks is extended to my wife, Goon Hi Lim, for her assistance in typing this material.

A DATA-FLOW ANALYSIS METHOD

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements of the Degree

Master of Science

---

by

Kwang Cook Lim

December 1976

ABSTRACT

Presented in this thesis is a method by which one can determine for a given program graph (1) what data definitions reach each node in the graph, (2) what data items have an upward exposed use at each node, and (3) what data definitions are "live" on each edge in the graph. The method is developed based on the topological sorting of edges and the acyclic image of a program graph. It is conceptually simple and easy to implement.

TABLE OF CONTENTS

		Page
CHAPTER I	INTRODUCTION	1
1.1	Application to Object Code Optimization	2
1.2	Application to Software Reliability	6
CHAPTER II	DATA-FLOW ANALYSIS OF LOOP-FREE PROGRAM	7
CHAPTER III	GENERAL DATA-FLOW ANALYSIS	19
3.1	Introduction	19
3.2	Basic Concepts	22
3.3	Analysis of Data Flow	26
3.4	Analysis of Data Use	32
CHAPTER IV	SUMMARY AND PROBLEMS FOR FUTURE RESEARCH	37
BIBLIOGRAPHY		39

CHAPTER I

INTRODUCTION

Presented in this thesis is a systematical method for extracting information concerning data-flow in a computer program.

The primary objective of data-flow analysis is to produce information that can be used in program optimization. In fact it has been applied to optimize object code in OS/360 FORTRAN H compiler. It has been shown that the optimization process using data-flow analysis can be bypassed to obtain a 40 percent reduction in compilation time. However, the use of this optimization process generally will result in a 25 percent reduction in the length of object codes and a two third reduction in execution time [7] .

Not only can data-flow analysis be utilized to optimize a program, but L.D. Fosdick and L.J. Osterweil [8] recently have shown that the technique can be used to detect certain types of anomalies in a program, and thus data-flow analysis can also be applied to improve software reliability.

Before we proceed to describe the data-flow analysis method, we shall use simple examples to illustrate possible application of data-flow analysis in the following.

1.1 APPLICATION TO OBJECT CODE OPTIMIZATION

To see how the technique of data-flow analysis can be utilized in object code optimization, let us consider the FORTRAN program shown in Fig. 1.1. This program computes the mean value AVG and the greatest common divisor LAG of integers X and Y.

```
      PROGRAM GCD (X,Y)
      IMPLICIT INTEGER (A-Z)
      LAG  =  X
      SML  =  Y
2     AVG  =  (X + Y) / 2
      IF(LAG.GE.SML) GO TO 4
      COPY =  LAG
      LAG  =  SML
      SML  =  COPY
      IF(SML.EQ.0) GO TO 5
4     LAG  =  LAG - SML
      GO TO 2
5     WRITE X,Y, AVG, LAG
      RETURN
      END
```

Fig. 1.1 Example Program

In order to optimize the object code of this program in compilation, we need to know where in the program the variables are defined and referenced. This can be accomplished by performing a data-flow analysis on the program.

The analysis can be facilitated by decomposing it into program blocks [4] and representing it as a program graph as shown in Fig. 1.2. Note that I/O statements have been deleted because we wish to emphasize optimization of the computational object codes at this point.

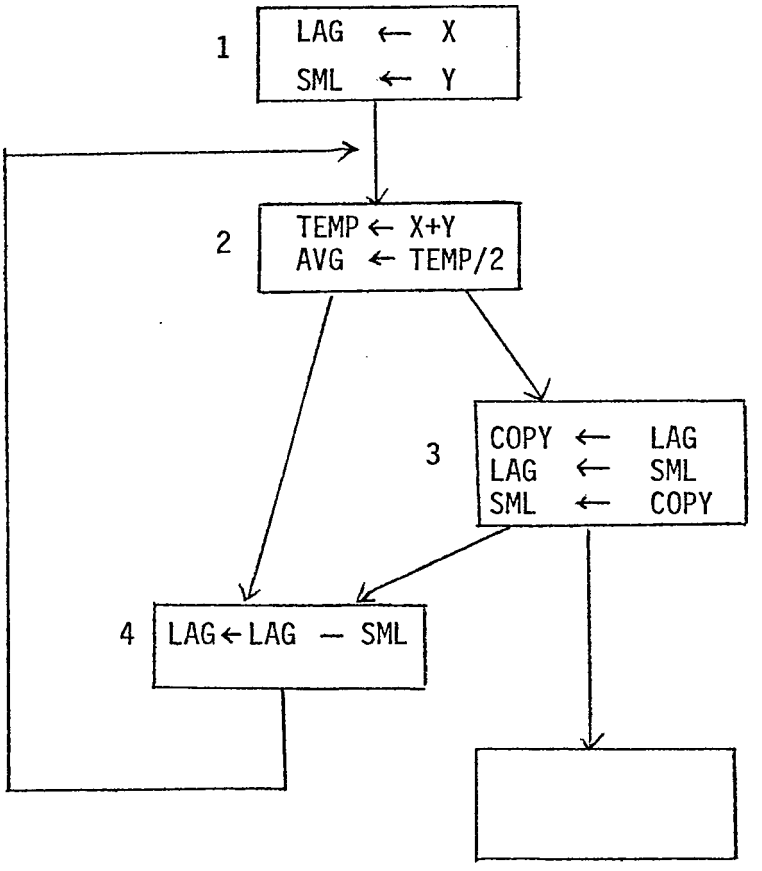


Fig. 1.2 Program graph GCD(X,Y) to be used for data-flow analysis.

A variable is called a data item in the data-flow analysis. A data definition is an expression or that part of an expression which modifies a data item. A data use is an expression which references a data item without modifying it. For example, an assignment statement of the form X←Y constitutes a data definition of the data item X and a data use of the data item Y. A data definition is said to be live on a path in the program if it is used subsequently. The heavy lines in Fig. 1.3 indicate the paths on which the definitions of LAG are live.

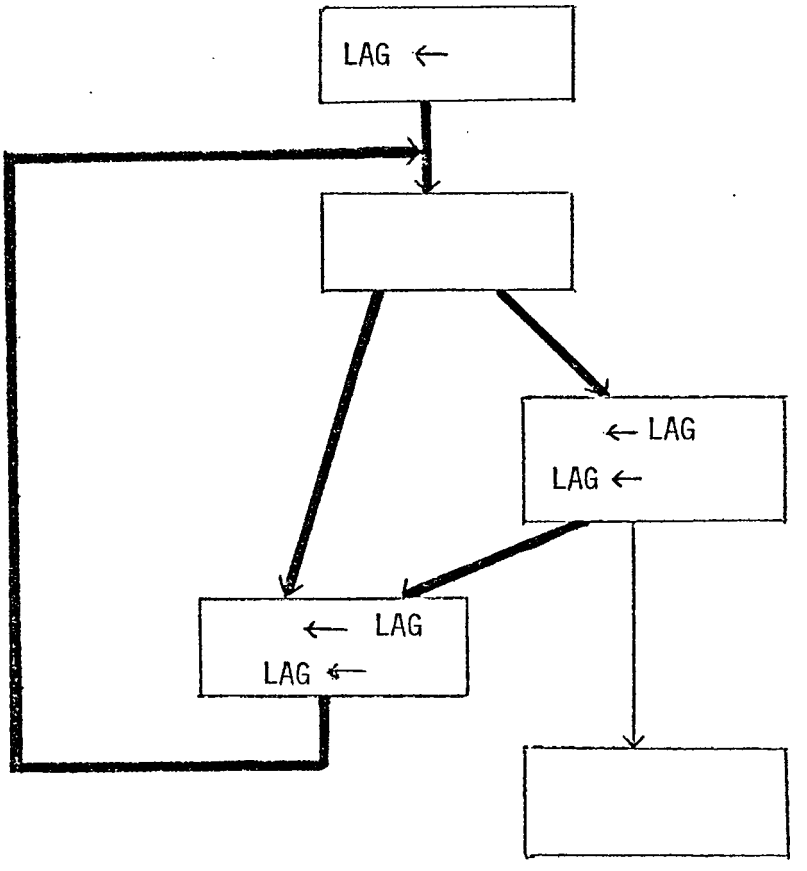


Fig. 1.3 Data flow of a data item LAG

The purpose of data-flow analysis is to determine what data definitions are available at any given point in a program graph, and where they are live or used. The information obtained can be used in object-code optimization as outlined below:

(1) Register Allocation

If a data item is assigned to a register and the data definition is live at the exit of a program block, then the corresponding store and load instructions may be omitted for obvious reasons.

(2) Loop Independent Computation

A data item is a constant in a loop if it is not defined anywhere in the loop. Any computation involving only such data items are said to be loop independent and can be moved out of the loop. For instant, TEMP←X + Y in Fig. 1.1 may be moved out of the loop if TEMP is not redefined elsewhere in the same loop. Note that if TEMP←X + Y moved backward out of the loop, then AVG←TEMP/2 also becomes loop independent, and thus can be moved backward out of the loop also. The loop independent computation can be identified by the result of a data-flow analysis.

(3) Subsumption

A data item X is said to be subsumed by a data item Y, if references to X can be replaced by references to Y. If X is subsumed by Y on a path,

then an expression of the form X←Y may be eliminated provided we replaced all references to X by a reference to Y. The relation of subsumption can be determined by means of data-flow analysis.

1.2 APPLICATION TO SOFTWARE RELIABILITY

Recently, Fosdick and Osterweil [8] suggest that data-flow analysis can also be used to detect data-flow anomalies in a program.

Data flow anomalies are caused by the abnormal pattern of data definitions and uses. For example, in Fig. 1.4, data item A is redefined in the second program block without any use of the first definition in the predecessor block. In other words, the first definition of A is not live on the path. The presence of a "dead" data definition in a program reflects a programming error, and can be detected by means of a data-flow analysis.

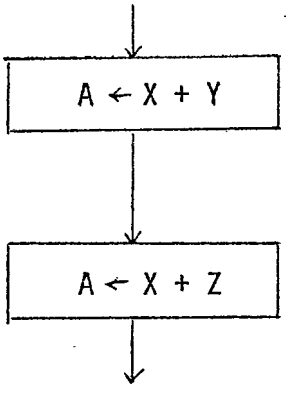


Fig. 1.4 Data Flow Anomaly

CHAPTER II

DATA-FLOW ANALYSIS OF LOOP-FREE PROGRAMS

In this chapter we shall discuss how to analyze data flow in a loop-free program.

For convenience, we shall be working with the control-flow graph of a program (or, program graph for short) in which each node corresponds to a basic program block [3,4,7] rather than a single statement. A basic (program) block is a linear sequence of program statements having the property that, if any statement in the sequence is executed, it entails that all other statements in the sequence are also executed. Each basic block will be labelled by a positive integer for reference. Since there is a one-to-one correspondence between a basic program block and a node in the control-flow graph, we shall use the words "block" and "node" interchangeably. Thus we may speak of block i being a successor block of block j if in the control-flow graph node i is a successor of node j in the graph-theoretical sense.

Three types of useful information may be derived from each basic block for the purpose of global data-flow analysis.

For any basic block i, the first type of useful information is a set, DB_i, of data definitions locally available in the block.

A locally available definition for a basic block is the last definition of a data item in the basic block. For example, $DB_i = \{X_i, Y_i\}$ for the basic block shown in Fig. 2.1. Note that a data definition is denoted by the name of the data item indexed by the block label. This notational convention will be used throughout this work.

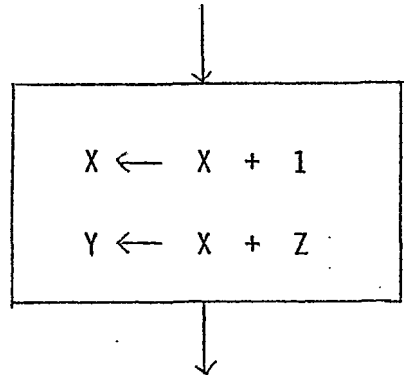


Fig. 2.1 The i^{th} program block

The second type of information that can be derived from a basic block is a set, UB_i, of data items having an exposed use locally. A data item is said to have a locally exposed use if it is used in the block and its use is not preceded by a definition in the block. For example, data item X has a locally exposed use in Fig. 2.1 because it appears on the right-hand side of the first assignment statement. Note, however, that its use in the right-hand side of the second assignment statement is not locally exposed because it is preceded by a definition of X (i.e., X←X + 1) in the block. Hence $UB_i = \{X, Z\}$ for the basic block shown in Fig. 2.1.

Finally, the third type of information that can be obtained from a basic block is a set, PB_i, of data items whose definitions are locally preserved. Definitions of a data item is said to be preserved through the block if that data item is not redefined in the block. Thus $PB_i = \{Z\}$ for the basic block shown in Fig. 2.1.

The three types of information described above can be utilized to analyze the flow of data definitions on a path in the control-flow graph.

A definition d in basic block i is said to reach basic block k if (1) d is a locally available definition from basic block i, (2) block k is a successor of block i, and (3) there is at least one path from block i to block k such that there is not a basic block on the path containing a redefinition of the same data item.

Any definition of a data item in the basic block is said to kill all previous definitions of the same data item. On the other hand, if a basic block does not contain a definition of a data item, then all previous definitions of this data item that reach the basic block are said to be preserved. Thus, for example, data definitions X_i and X_j (i.e., $X \leftarrow 1$ and $X \leftarrow 2$, respectively) in Fig. 2.2 are killed by the redefinition X_k (i.e., $X \leftarrow 3$) while data definitions X_i

and X_j in Fig. 2.3 are preserved through basic block k.

Fig 2.2

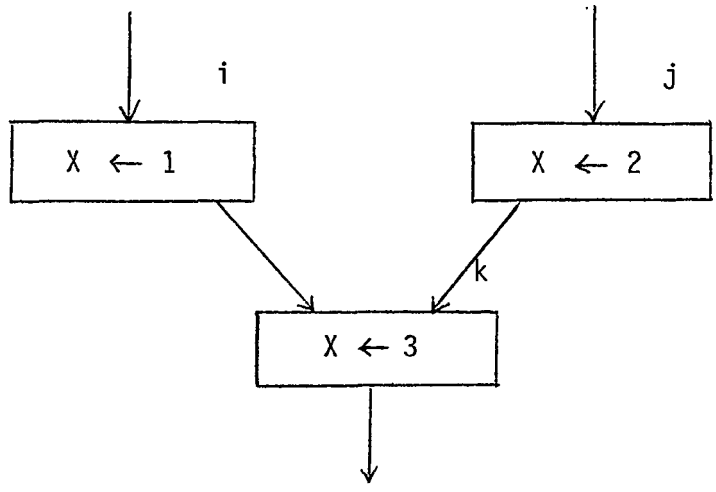
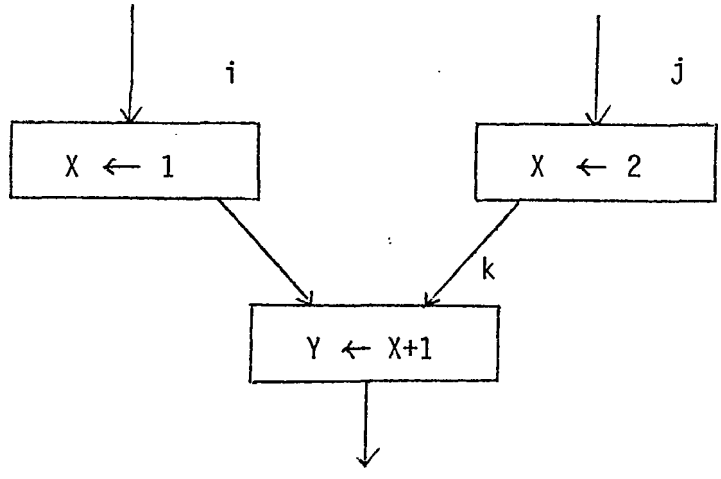


Fig 2.3



For each node in the control-flow graph of a program, we are interested in knowing what data definitions reach that node and what data definitions are available from that node.

Let R_i denote the set of data definitions that reach node i, and let A_i denote the set of data definitions available from node i. Evidently, all data definitions available from the immediate predecessors of node i reach that node. That is,

$$R_i = \bigcup_p A_p \tag{2.1}$$

for all node p which is an immediate predecessor of node i. The question now is ; how do we compute the set of data definitions available from a node to its immediate successors ? Obviously, by our definition, any data definition locally available in node i is also available to its immediate successors. In addition, any data definition that reaches node i and is preserved through the node is also available to its immediate successors. Thus

$$A_i = RP_i \cup DB_i \tag{2.2}$$

where

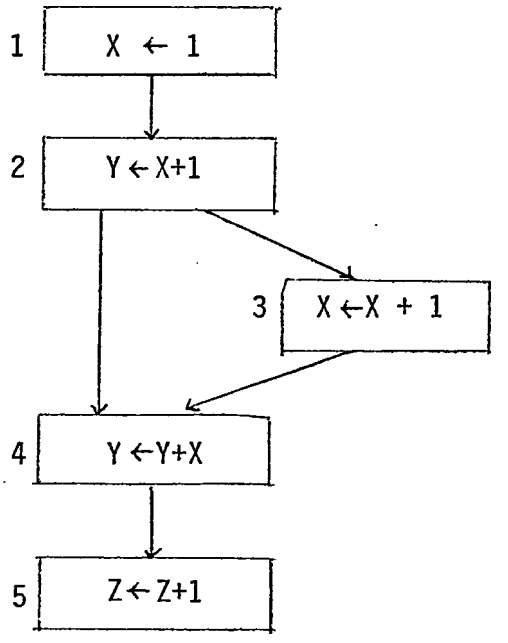
$$RP_i = \{ d \mid d \in R_i \text{ and } d \text{ is a definition of data item } X \text{ and } X \in PB_i \} \tag{2.3}$$

The sets R_i and A_i defined above characterize the forward data flow in the control-flow graph of a program. What we would

like to accomplish here, among other things, is to devise an effective method to compute the membership of R_i and A_i for each node i in terms of DB_i and PB_i.

If node i is an entry node, then $R_i = \emptyset$ since an entry node does not have any predecessor. Consequently, by (2.2), $A_i = DB_i$. This result can then be used to compute the sets associated with its immediate successors. And by repeating the same process, we can compute R_i and A_i for all nodes in the graph. To illustrate, we compute the forward data flow of the program shown in Fig 2.4 as listed in Table 2.1.

Fig 2.4



Tab. 2.1 Forward data flow by set representation

Node	DB_i	PB_i	R_i	A_i
1	X_1	Y, Z	$\emptyset$	X_1
2	Y_2	X, Z	X_1	X_1, Y_2
3	X_3	Y, Z	X_1, Y_2	Y_2, X_3
4	Y_4	X, Z	X_1, X_3, Y_2	X_1, X_3, X_4
5	Z_5	X, Y	X_1, X_3, Y_4	X_1, X_3, Y_4, X_5

If the method is to be implemented by using a computer, then the computations involved can be greatly facilitated by encoding all quantities involved as follows. We shall represent a set by a binary string of length equal to the total number of data definitions in the program to be analyzed. Each bit position corresponds to a particular data definition. For a set of data definitions, the i-th bit in its binary-string representation will be equal to 1 if and only if it contains the i-th data definition as its member; otherwise it is equal to 0. For a set od data items, if it contains a data item, say, X as its

member, then all bit positions in its bit-string representation corresponding to a definition of X will be equal to 1. Otherwise they will be equal to 0.

By using the above coding scheme, we can readily compute R_i and A_i as indicated below :

$$R_i = \bigoplus_{j \in P} A_j \quad P \text{ is the set of immediate predecessor of } i \tag{2.4}$$

$$RP_i = R_i \odot PB_i \tag{2.5}$$

$$A_i = RP_i \oplus DB_i, \tag{2.6}$$

where $\odot$ and $\oplus$ denote bit-wise AND and OR operations, respectively. Thus the forward data flow of the program shown in Fig. 2.4 can be computed as illustrated in Tables 2.2 and 2.3.

Table 2.2 Bit Position Table

Position	Definition
1	X_1
2	X_3
3	Y_2
4	Y_4
5	Z_5

Table 2.3 Bit string representation of forward data flow

Node	DB_i	PB_i	R_i	A_i
1	10000	00111	00000	10000
2	00100	11001	10000	10100
3	01000	00111	10100	01100
4	00010	11001	11100	11010
5	00001	11110	11010	11011

A data item is said to have an upward exposed use in basic block i if either it has a locally exposed use in the block (cf. page 8) or there exists a path from node i to node k in the program graph such that (1) the data item has a locally exposed use in node k, and (2) there does not exist a node on the path that contains a definition of that data item.

The set of data items having an upward exposed use in block i is denoted by U_i, and can be computed by scanning backward along the paths leading from node i. Specifically, for the exit node n, we have

$$U_n = UB_n \quad . \tag{2.7}$$

and for any other node i,

$$U_i = (\bigcup_{j \in S} U_j \cap PB_i) \cup UB_i, \qquad (2.8)$$

where S is the set of immediate successors of i.

Set U_i characterizes the backward flow of data, and can be represented by a binary string as described previously. Formula (2.8) can be rewritten as shown below :

$$U_i = (\bigoplus_{j \in S} U_j \odot PB_i) \oplus UB_i \qquad (2.9)$$

if all sets involved are given in the binary representation. The backward data flow of the program given in Fig. 2.4 is shown in Table 2.4 for illustration

Table 2.4 Backward data flow in binary representation

Node	UB_i	PB_i	U_i
1	00000	00111	00001
2	11000	11001	11001
3	11000	11001	11111
4	11110	11001	11111
5	00001	11110	00001

It is natural for us to say that a data definition d of a data item X is <u>live</u> at basic block i if d reaches this node and X has an upward exposed use at the node. Hence L_i, the set of data definitions live at node i, can be formally defined as

$$L_i = \{ d \mid d \in R_i \text{ and } d \text{ is a definition of data item } X \text{ and } X \in U_i \}$$

In binary-string representation L_i can be readily computed by using the following relation :

$$L_i = R_i \odot U_i \tag{2.10}$$

In global data-flow analysis it is more convenient to consider live definition along a path rather than at a node. For that purpose, we need to consider L_e, the set of data definitions that are live along an edge e. Formally, for an edge e that enamates from node p and terminates at node s .

$$L_e = \{ d \mid d \in A_p \text{ and } d \text{ is a definition of data item } X \text{ and } X \in U_s \}.$$

The membership of L_e in binary-string representation can be computed as follows :

$$L_e = A_p \odot U_s \qquad (2.11)$$

To illustrate, we shall compute, based on the information contained in Tables 2.3 and 2.4, the live data definitions on each edge in Fig. 2.4 as shown in Table 2.5.

Table 2.5 Live - Dead Analysis

Edge	A_p	U_s	L_e
1 2	10000	11001	10000
2 3	10100	11111	10100
3 4	01100	11111	01100
2 4	10100	11111	10100
4 5	11010	00001	00000

It is obvious from (2.10) and (2.11) that, to perform live-dead analysis, we need to compute the membership of R_i, A_i, and U_i for each node i in the program graph. As demonstrated above, if the program to be analyzed is loop-free, then R_i and A_i can be computed by scanning forward a path in the program graph while U_i can be computed by scanning a path backward. However, if the program to be analyzed contains a loop, then the problem become more complicated. This is the subject of discussion in the next chapter.

CHAPTER III

GENERAL DATA-FLOW ANALYSIS

3.1. INTRODUCTION

In the preceding chapter we have shown that equations (2.4), (2.5), and (2.6) can be used to determine for each node in the program graph the data-definitions reaching that node. Note that we can accomplish this in a single pass through the graph if we can order the nodes in such a way that a node will not be visited until all of its predecessors are visited. This ordering can always be obtained for a loop-free program graph by performing a topological sort of the nodes [6]. However, this will not be the case for a program containing a loop.

To fix the idea, let us consider the program graph shown in Fig. 3.1.

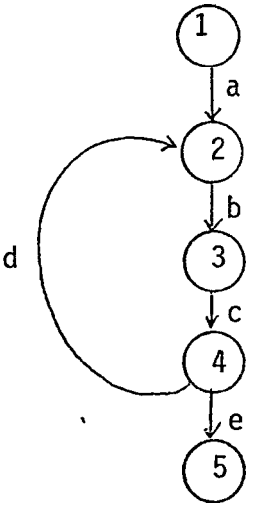


Fig. 3.1 A program graph with a loop

Obviously, the set of data definitions reaching node 2, the entry node of the loop, is the union of the set of definitions reaching through path a and that reaching through path abcd. Thus the data definitions reaching nodes 2,3 and 4 can not be completely determined until the second pass through these nodes. In general, a node may have to be traversed more than twice if it is a part of a nested loop structure.

A natural solution to this problem would be first to compute R_2', the set of definitions reaching node 2 through path a. We then proceed to compute R_2'', the set of definitions reaching node 2 through path abcd by using the method described in the preceding chapter. We can then set (or, initialize) the value of R_2, the true set of definitions reaching node 2, to be the union of R_2' and R_2'', i.e., $R_2 = R_2' \cup R_2''$. Once this is done, we can delete edge d to obtain a loop-free representation of the program. The definitions reaching nodes 3,4,and 5 can now be determined by using the method described in the preceding chapter.

If the program to be analyzed contains nested loops, then the process described above can be repeatedly applied (inner loop first) until a completely loop-free representation is obtained.

To effectively carry out the process outlined above, we need

to be able to (1) identify a node being the entry of a loop. (such as node 2 in Fig. 3.1), and (2) find an ordering of the nodes that constitute the loop structure so that the method described in the preceding chapter can be applied to determine the data definitions reaching the entry node through the loop structure. These two problems are not difficult in principle. The question here is : how can we do it with a minimal cost ?

A minimal-cost solution to the first problem has been suggested by Lowry and Medlock [7] by using the so-called predominance relation defined on the nodes of a graph. The second problem can be solved by using the interval analysis suggested by Allen and Cocke [3].

In the following we describe yet another method that can be used to solved both problems simultaneously with a minimal cost. Our method is believed to be more efficient than the methods of predominance relation and interval analysis in that our method does not require a rather inefficient search through the graph while the other two do at certain stages in their applications.

3.2 BASIC CONCEPTS

It is observed that, to use equations (2.4),(2.5), and (2.6) to compute the forward data flow on an edge (i,j) in the program graph, we need first to compute the definitions that reach node i through each and every edge terminating at node i. For this reason it is useful to order the edges by using the predominance relation.

Let x and y be two edges in the program graph. Edge x is said to predominate edge y if and only if, in a traversal of any path leading from the entry node that includes edges x and y, the first pass through edge x must precede the first pass through edge y. For example, in Fig. 3.2, edge a predominates edges b and c, edge c predominate edges d and e, and edge f predominates edge g.

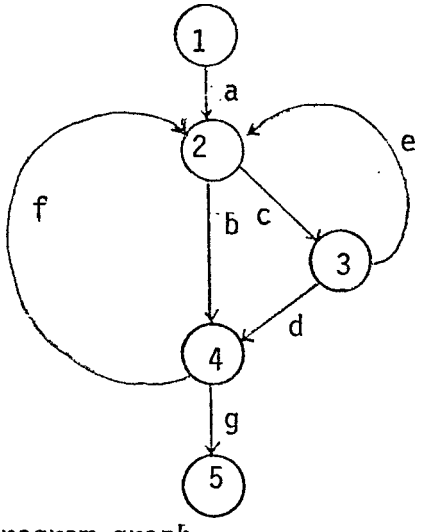


Fig. 3.2 A program graph.

Note that the predominance relation defined above is different from that defined by Medlock [7] in that it is a binary relation defined on the set of edges rather than nodes.

It is easy to verify that the predominance relation is irreflexive, asymmetric, and transitive, and therefore is a partial ordering [6] on the set of edges. Consequently, we can use this relation to sort the edges in a program graph topologically. For example, we can topologically sort the edges in Fig. 3.2 to yield the following ordering :

abcdefg

Next, we observe that in program optimization we will be most interested in the data items whose definitions remain to be the same while a loop is being iterated. For this reason, when we come to initialize the entry node of a loop, we need only to go through the loop structure exactly once. With this assumption we can greatly simplify the process of data-flow analysis by considering the acyclic representation of a (cyclic) graph.

Given a graph G', we can construct another graph G from G', as follows :

(1) G contains all nodes in G',

(2) An edge (i,j) in G' is also an edge in G if and only if node

j is not on any path from the entry node to node i in G'.

(3) If there is an edge (i,j) in G' such that node j is on some path from the entry node to node i, then add a copy of node j to G, and let edge (i,j) be in G leading from node i to that copy of node j.

It is easy to verify that graph G constructed as described above is acyclic. Hence we shall refer to G as the acyclic image of G' throughout this work.

Figure 3.3 shows the acyclic image of the program graph given in Fig. 3.2. Note that a copy node is identified by a double circle in Fig. 3.3.

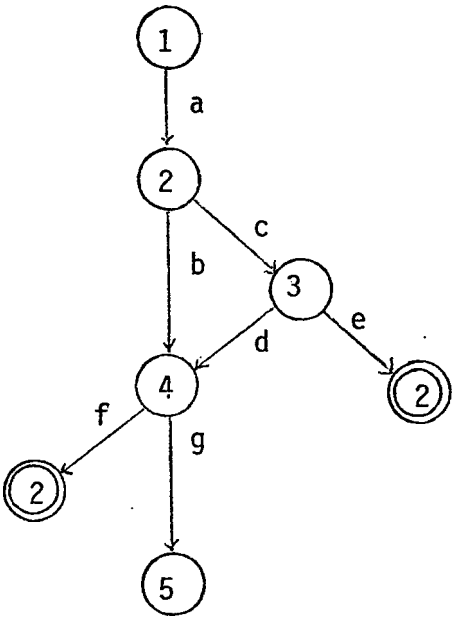


Fig. 3.3 The acyclic image of the graph shown in Fig. 3.2.

In constructing the acyclic image of a program graph we need to add a copy node only if there is a loop in the graph. Furthermore, a node for which we need to produce a copy is the entry node of a loop. If a program graph is loop-free then its acyclic image is identical to the original program graph. If a program graph contains a loop, then there is a copy of the loop-entry node for each path that constitutes the loop structure. Therefore, a node that has a copy in the acyclic image is the one that has to be initialized before data-flow information can be determined globally.

The significance of a copy node in the acyclic image of a program graph is that we can use equations (2.4),(2.5), and (2.6) to systematically compute the definitions reaching that node. Specifically, if node i has k copies ($k \geq 1$) in the acyclic image of the program graph, then we can apply the three equations to compute R_i^o, the set of definitions that reaches node i when it is entered for the first time, and $R_i^{(j)}$, the set of definitions reaching the j-th copy of node i, for all $1 \leq j \leq k$. R_i,the set of definitions reaching node i, can then be properly initialized to $R_i = R_i^{(o)} \cup R_i' \cup R_i'' \cup \ldots \cup R_i^{(k)}$.

Once this is done, the k copies of node i in the acyclic image of the program graph will no longer be of any use and thus can be deleted. Now if we repeat the above process to initialize all loop-entry nodes in the acyclic image of the program graph, we will be left with an acyclic graph without copy nodes. Based on this acyclic graph we can

apply the technique described in the preceding chapter.to perform data-flow analysis.

3.3 ANALYSIS OF DATA FLOW

It should be obvious now that when we work with the acyclic image of a program graph, the computation involved in initializing a loop-entry node is exactly the same as that involved in analyzing the forward data flow of a loop-free program. We sould note, however, that while we can compute R_i and A_i for each node in a single pass through the graph, we can not initialize all loop-entry nodes in the same pass. To initialize a loop-entry node, we have to start the computation from the program entry. The computation is completed when the values of R_i and A_i for all copies of the loop-entry node in question are determined.

Having explained the basic strategy, we can now state our algorithm for forward data-flow analysis as follows : given a program graph G',

1. Topologically sort the edges in G' in accordance with the predominance relation to order the edges in the program graph.
2. Construct G, the acyclic image of G'.
3. Initialize $R_i \leftarrow \emptyset$ and $A_i \leftarrow DB_i$ for each and every node in G.
4. Set (i,j) to be equal to the first edge in G.

5. Set

   $R_j \leftarrow R_j \oplus A_i$ and then

   $A_j \leftarrow A_j \oplus (R_j \odot PB_j)$

   (all sets in binary - string representation)

6. If (i,j) is the last edge in G, the algorithm terminates.

7. If (i,j) terminates at a copy node in G, set c = j and delete this copy node from G. Otherwise set (i,j) to next edge (in the order obtained in step 1) and go to step 5.

8. If there exists another copy of node c in G, then set (i,j) to next edge (in the order obtained in step 1) and go to step 5. Otherwise go to step 4.

When the above algorithm terminates, we will be left with an acyclic graph without copy nodes, and all nodes are associated with proper values of R_i and A_i. Thus we have obtained complete information on the flow of data definitions in the program graph.

To illustrate the idea involved, let us consider the program graph show in Fig. 3.4.

Note that the graphical structure of this program graph is identical to that of Fig. 3.2. Thus by topologically sorting the edge as described in Sec. 3.2 yield the following ordering of edges:

a,b,c,d,e,f,g,

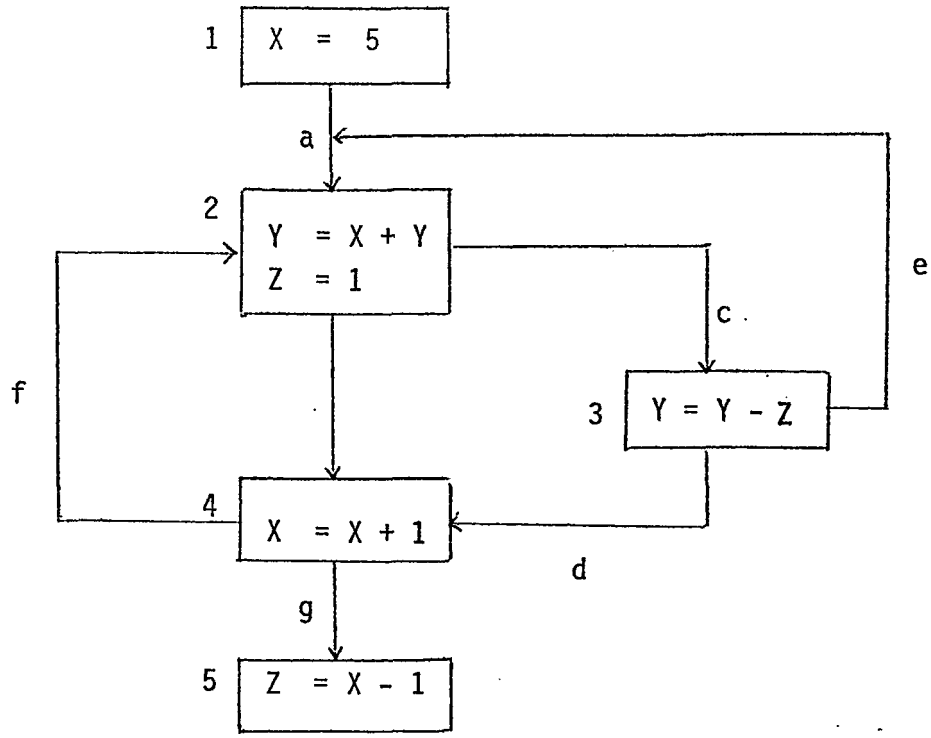


Fig. 3.4 A program graph

or

(1,2),(2,4),(2,3),(3,4),(3,2),(4,2),(4,5)

Next, we construct the acyclic image of the program graph as shown in Fig. 3.5.

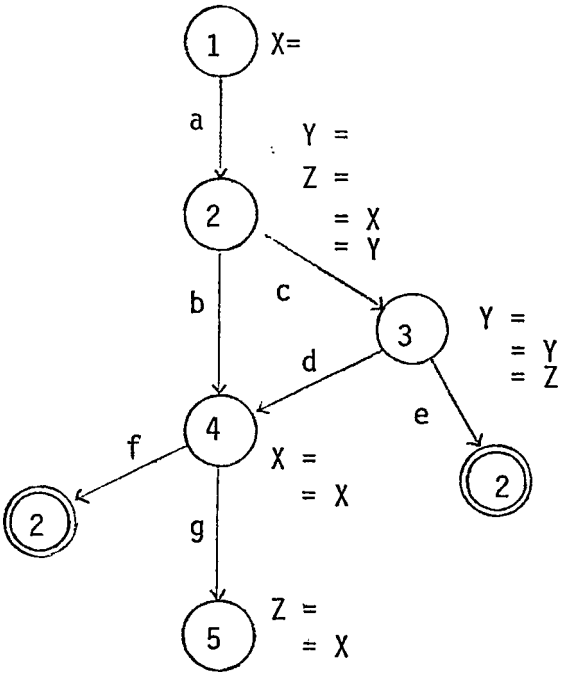


Fig. 3.5 The acyclic image of Fig. 3.4.

Now if we encode the sets involved into binary strings in the sequence :

$$X_1Y_2Z_2Y_3X_4Z_5,$$

then the process of applying the algorithm to the program graph can be readily traced in Table 3.1. The last entries in each column contain the desired values of R_i and A_i for the corresponding node. For instance, the forward data flow at node 5 is characterized by R_5 = 011110 and A_5 = 010111.

The values of PB_j, which remain constant throughout the process, are included in the table to facilitate computation.

Table 3.1 Information obtained by applying the forward data-flow analysis algorithm to Fig. 3.4.

	Node / initial values / after edge(i,j) is processed		1	2	3	4	5
	initial values	R_j	000000	000000	000000	000000	000000
		PB_j	011101	100010	101011	011101	110110
		A_j	100000	011000	000100	000010	000001
First Pass	a (1,2)	R_j		100000			
		PB_j		100010			
		A_j		111000			
	b (2,4)	R_j				111000	
		PB_j				011101	
		A_j				011010	
	c (2,3)	R_j			111000		
		PB_j			101011		
		A_j			101100		
	d (3,4)	R_j				111100	
		PB_j				011101	
		A_j				011110	
	e (3,2)	R_j		101100			
		PB_j		100010			
		A_j		111000			
	f (4,2)	R_j		111110			
		PB_j		100010			
		A_j		111010			

Second Pass	a (1,2)	R_j PB_j A_j		111110 100010 111010			
	b (2,4)	R_j PB_j A_j				111110 011101 011110	
	c (2,3)	R_j PB_j A_j			111010 101011 101110		
	d (3,4)	R_j PB_j A_j				111110 011101 011110	
	g (4,5)	R_j PB_j A_j					011110 110110 010111

3.4 ANALYSIS OF DATA USE

Since an upward exposed use of a data item propagates backward along a path, we can compute the membership of U_i, the set of data items having an upward exposed use at node i, only after we have computed the same for all of its successor nodes. This can be accomplished by processing the edges in the order opposite to that used in the forward data-flow analysis. Again, the loop-entry nodes have to be initialized before a global analysis can be performed. The computation involved in initializing the copy nodes is similar to that involved in the forward data-flow analysis except that it will be carried out in the reverse order.

Our algorithm for backward data-flow analysis can be stated as follows : given a program graph G',

1. Obtain an ordering of the edges in G' by first topologically sorting the edges using the predominance relation, and then reversing the order so obtained.
2. Construct G, the acyclic image of G'.
3. Set $U_i = UB_i$ for each and every node in G.
4. Set (i,j) to be equal to the first edge (in G in the order obtain in Step 1).

5. Set $U_i \leftarrow U_i \oplus (U_j \odot PB_i)$.
6. If (i,j) is the last edge in G, the algorithm terminates.
7. If there does not exist a copy of node i then set (i,j) to next edge (in the order obtained in step 1) and go to step 5. Otherwise set (k,i) to be equal to the first edge that terminates at a copy of node i.
8. Set $U_k \leftarrow U_k \oplus (U_i \odot PB_k)$ and delete this copy of node i as well as edge (k,i).
9. If there exists another edge (k,i) that terminates at a copy of node i, go to step 8. Otherwise go to step 4.

To illustrate the idea involved, we shall now apply this algorithm to the program graph given in Fig. 3.4 to determine U_i, the set of data items having an upward exposed use at node i for all $1 \leq i \leq 5$. We shall use the same scheme to encode the sets involved. The edges are to be processed in the order :

gfedcba.

The results are shown in Table 3.2. The desired value of U_i in to be found in the last entry of the corresponding column. For example, U_3 =111111 and U_1 =010100 . Again, the values of PB_i are unchanged and are included in the table for the purpose of facilitating the computation.

Table 3.2 Information obtained by applying the backward data-flow analysis algorithm to Fig. 3.4.

node / initial values / values after edge (i,j) is processed		1	2	3	4	5
initial values	U_i	000000	110110	011101	100010	100010
	PB_i	011101	100010	101011	011101	110110
g (4,5)	U_i				100010	
	PB_i				011101	
f (4,2)	U_i				110110	
	PB_i				011101	
e (3,2)	U_i			111111		
	PB_i			101011		
d (3,4)	U_i			111111		
	PB_i			101011		
c (2,3)	U_i		110110			
	PB_i		100010			
b (2,4)	U_i		110110			
	PB_i		100010			

	f (4,2) U_i / PB_i				110110 011101	
	e (3,2) U_i / PB_i			111111 101011		
	g (4,5) U_i / PB_i				110110 011101	
	d (3,4) U_i / PB_i			111111 101011		
	c (2,3) U_i / PB_i		110110 100010			
	b (2,4) U_i / PB_i		110110 100010			
	a (1,2) U_i / PB_i	010100 011101				

We shall conclude this section by using the results contained in Tables 3.1 and 3.2 to compute the set of live data definitions along each edge in Fig. 3.4 as shown in Table 3.3. Recall that $L_{(i,j)}$, the set of definition live on edge (i,j), can be computed by the formula:

$$L_{(i,j)} = A_i \odot U_j$$

as discussed previously (cf. (2.11)).

Table 3.3 The results of a live-dead analysis on Fig. 3.4

Edge (i,j)	A_i (from Table 3.1)	U_i (from Table 3.2)	$L_{(i,j)}$
(1,2)	100000	110110	X_1 100000
(2,4)	111010	110110	X_1, Y_2, X_4 110010
(2,3)	111010	111111	X_1, Y_2, Z_2, X_4 111010
(3,4)	10110	110110	X_1, Y_3, X_4 100110
(3,2)	101110	110110	X_1, Y_3, X_4 100110
(4,2)	011110	110110	Y_2, Y_3, X_4 010110
(4,5)	011110	100010	X_4 000010

CHAPTER IV

SUMMARY AND PROBLEMS FOR FUTURE RESEARCH

In this thesis we have presented a method by which we can determine for a given program graph,

1. What data definitions reach each node in the graph.
2. What data items have an upward exposed use at each node.
3. What data definitions are "live" on each edge in the graph.

The method is developed based on the topological sorting of edges and the acyclic image of a program graph. It is conceptually much simpler than all known methods [3,7,8], and is thus more cost effective to implement. This is important because, as we have mentioned previously, the problems of data-flow analysis are not fundamentally difficult. The main interest is in the method that is simple and thus easy to implement.

The practical value of this method can be greatly enhanced if we further research into the problems of how to implement it on a computer with a minimal cost. For instance, what information structures are to be used to achieve our goal? What features of a programming language (in which the program to be analyzed is written)

can be exploited to effectively construct the program graph and its acyclic image? These are important problems that need to be dealt with in the future.

BIBLIOGRAPHY

BIBLIOGRAPHY

* 1. Aho, A.V. and Ullman, J.D. The Theory of Parsing Translation, and Compiling, Vol. 1, Pritice - Hall, Englewood Cliffs, N. J., 1973.

* 2. Aho, A.V. and Johnson, S.C., "Deterministic Parsing of Ambiguous Grammar", CASM, Vol. 18, No. 8, pp441-452, August, 1975.

3. Allen, F.E. and Cocke, J., "A program Data Flow Analysis Procedure" CACM, Vol. 19, No. 3, pp 137-147, March, 1976.

4. Bauer, F.L., Compiler Construction, springer - Verlay Berlin, Heidelberg, 1974.

* 5. Elson, Mark, Concepts of programming Languages, Science Research Associations, Chicago, 1973.

6. Knuth, D.E., The Art of Computer Programming, Vol. 1, Fundamental Algorithm, Second Edition, Addison Wesley, Reading, Mass., 1973.

7. Lowry, Edward S, and Medlock, C.W., "Object Code Optimization", CACM, Vol. 12, No. 1, pp 13-22, January, 1969.

8. Osterweil, L.J. and Fosdick, L.D., "Data Flow Analysis in Software Reliability", Department of Computer Science Technical Report #CU-CS-087-76, University of Colorado, Boulder, Colorado, May 1976.

* References not cited in the text.