DATA FLOW ANALYSIS

Juan-Jose Barroso

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

December 1982

ABSTRACT

DATA FLOW ANALYSIS

Juan-Jose Barroso

Data Flow Analysis techniques are being used by the compilers to optimize the object code. This report presents a systematic survey of Data Flow Analysis for code optimization in compilers. The different methods used to gather program data flow information are discussed and a comparison of the performance and easiness of implementation of the different data flow analysis techniques is also presented. Some of the new areas of applications of data flow analysis techniques are listed.

ACKNOWLEDGMENTS

I want to thank my supervisor Dr. Opatrny for his technical advice and his assistance during the preparation and the revision of the manuscript.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

1.1 Motivation

A compiler translates source programs usually written in a high level language to object programs in a low level language. The object code generated by a straighforward code generation in a compiler is usually inefficient and an optimization phase is required during the compiling process to improve the code. The main objective of optimization is to increase the run-time efficiency of the generated programs while preserving the program equivalence. We should note that by optimization of the code we don't mean finding of the best program since there could be no best program, but finding of an improved equivalent one [Blum67,Alle81].

Although some improvement can be done simply by scanning the program as, for example, constant value substitution, most of the optimizations require a knowledge of the definition and use of the different variables throughout a program or other relevant information to a particular optimization. Gathering the necessary

information is known as data flow analysis which is the central topic of this study.

Data flow analysis consists of determining static characteristics of a program and it is usually performed on some intermediate form of the compiled program such as "three address" statements. Data flow analysis assumes that all paths of the program control flow can represent actual execution even though it is not always the case. The result of data flow analysis is an approximate description of the static characteristics of a program. Two reasons can be given, however, for the use of this method [Char81]. The first one is that distinguishing the paths which represent actual execution from those which do not is in general an undecidable problem. The second reason is that such an approach allows simple algorithms and simple data structures to gather data flow information.

Code optimization was already a preoccupation when the first compilers were designed in 1950's. The scarce central memory and the low speed of the computers of that period explains the need of code optimization. Following the account of Backus [Back81], the FORTRAN compiler which was written from 1954 to 1957, implemented several optimizing transformations. Common subexpressions elimination, motion of code out of the loops and register allocation were among such transformations. An analysis of the data flow of the program was performed by the compiler to allow these transformations. Code optimization received additional

attention during the 1960's when several optimizing compilers were produced (PL/I, ALGOL, FORTRAN). It became apparent that some optimization techniques are applicable to several programming languages and that they require the knowledge of data flow in a program. Thus, general principles of optimization techniques and the related data flow analysis started to be investigated as a separate topic. Such a systematic understanding of the optimization methods made them generally applicable to any compiler. Today the literature on code optimization and data flow analysis is quite extensive. An overview of some techniques of data flow analysis can be found in [Hech77,Aho77b,Kenn81].

In this report we will present a systematic survey of data flow analysis and a comparison of the performace and easiness of implementation of the different data flow analysis techniques. As part of this introductory chapter, a summary of the main types of optimizations done by compilers are presented. A brief discussion of control flow concepts follows. We then list typical known data flow analysis problems which arise in the optimization process. Finally, we name the different methods used to solve data flow analysis problems.

In Chapters 2 and 3 we present the traditional methods of data flow analysis, namely iterative and interval analysis. Although these methods can be considered as particular cases of the more general method to be presented

in Chapter 4, their explanation will make the understanding of the general method easier.

In Chapter 4 we present data flow analysis frameworks which constitute a unified approach to data flow analysis. The known data flow analysis problems can be modeled by this general method. Chapter 4 is therefore central to this report.

Interprocedural and high level data flow analysis is introduced in Chapter 5.

In the last Chapter we will give a comparison of data flow analysis algorithms along with some concluding remarks.

1.2 Program Optimization.

Several transformations can be done to improve the execution time of compiled programs. A survey of such transformations can be found in [Alle72] and a more concise enumeration in [Kenn81]. Those program transformations can be divided in two main categories: local, which consider only straight lines of code without branching, and non-local transformations in which the knowledge of control flow of a program is required. The topic of this report being data flow analysis, we will be mainly interested in non-local transformations. These non-local transformations can be divided into three categories: global, loop, and machine dependent transformations. We define now the most typical program transformations in these three categories.

(i) Global transformations.

(a) Redundant subexpression elimination. An expression A **op** B may be eliminated if its value is already available, that is, if the expression A **op** B has been previously computed and the operands A or B have not been redefined (their value has not been changed).

(b) Constant propagation. This optimization replaces expressions by their value when all the operands are constants or names whose values are fixed. These expressions may be evaluated at compile time.

(c) Dead code elimination. As a result of constant propagation, some instructions may become "dead" because their results are never used or they cannot be reached. Dead code elimination detects and deletes such instructions.

(d) Variable propagation. Instructions of the form A:=B may be eliminated if the subsequent uses of A can be replaced by B.

(e) Procedure integration. This optimization is performed at the subprogram linkage level. In an "open-linkage" the called routine replaces the call statement in the caller. In a "semi-open linkage" the called routine is compiled together with its caller which avoids the overhead associated with the standard linkage (close linkage), where no information of the called routine is available during the compilation of the caller.

(ii) Loop transformations.

(a) Code motion. Expressions whose value doesn't change inside a loop may be moved out of the loop to avoid computing them at each loop iteration.

(b) Strength reduction. This transformation replaces slow operations by faster operations. This may be done in instructions that depend on the the variable used for loop iteration (induction variable). For example, if the induction variable varies linearly, a reference to an array element could be computed by the addition of a constant to the previous reference address instead of using the general formula to compute an array address which includes a multiplication.

(c) Linear test replacement. After performing strength reduction, the only use of the induction variable is frequently in the test which controls the loop iterations. Linear test replacement replaces such induction variable by an induced temporary variable. Instructions involving the induction variable become useless and they will be deleted by dead code elimination.

(iii) Machine dependent transformations.

(a) Register allocation. This optimization tries to make efficient use of the registers by deciding which variables should reside in registers and to which register

each variable should be assigned. Optimized register allocation can eliminate some load and store instructions.

(b) Instruction scheduling. Instructions can be scheduled to take advantage of the machine architecture to improve the execution time. This can be very important in computers with pipelined arithmetic units.

(c) Storage mapping. This transformation seeks to reduce the amount of active memory space used by a program by reusing the space occupied by variables whose value is not anymore needed and, in computers with paging, it tries to map together the code which is going to be used at the same time.

(d) Detection of parallelism. For machines with multiple units or vector machines, it is desirable to detect operations which may be executed in parallel.

1.3 Control Flow Analysis.

In general, data flow analysis of a program is preceded by the "control flow analysis". The purpose of control flow analysis is to subdivide the program into "basic blocks" and to find all possible transfers from one block to another block. A "basic block" is a straight line of statements with only one entry (top) and one exit (bottom) and which will be executed in sequence. The control flow of a program may be represented by a graph whose nodes are basic blocks and the edges represent possible transfers between nodes.

The resulting graph is called the "flow graph".

Formally, a flow graph [Alle70,Aho76] is a triple G = (N,E,s) where

(i) N is a finite set of nodes.

(ii) E is a set of edges which is represented by pairs of nodes connected by them. An edge (x,y) is from the node x to the node y and we say that x is a "predecessor" of y and y is a "successor" of x.

(iii) s is the initial node and there is a path from s to every node x in N.

(iv) A path of G is a sequence x1, x2, ..., xk such that (xi, xi+1) $\in$ E for 1<=i<k.

Figure 1.1 ilustrates the basic blocks of a Pascal-like program and Figure 1.2 shows the flow graph of such a program.

For most program optimizations local data flow information is first obtained at each node of its flow graph and then this information is propagated through the flow graph. Data flow inside each basic block can be represented by a directed acyclic graph (DAG) [Aho77b], and local information can be obtained by an anlysis of the DAG. A DAG is a directed graph with no cycles which shows how the values computed by each statement in a basic block are used by the subsequent statements in the basic block. Considering a three address statement code, the DAG of a

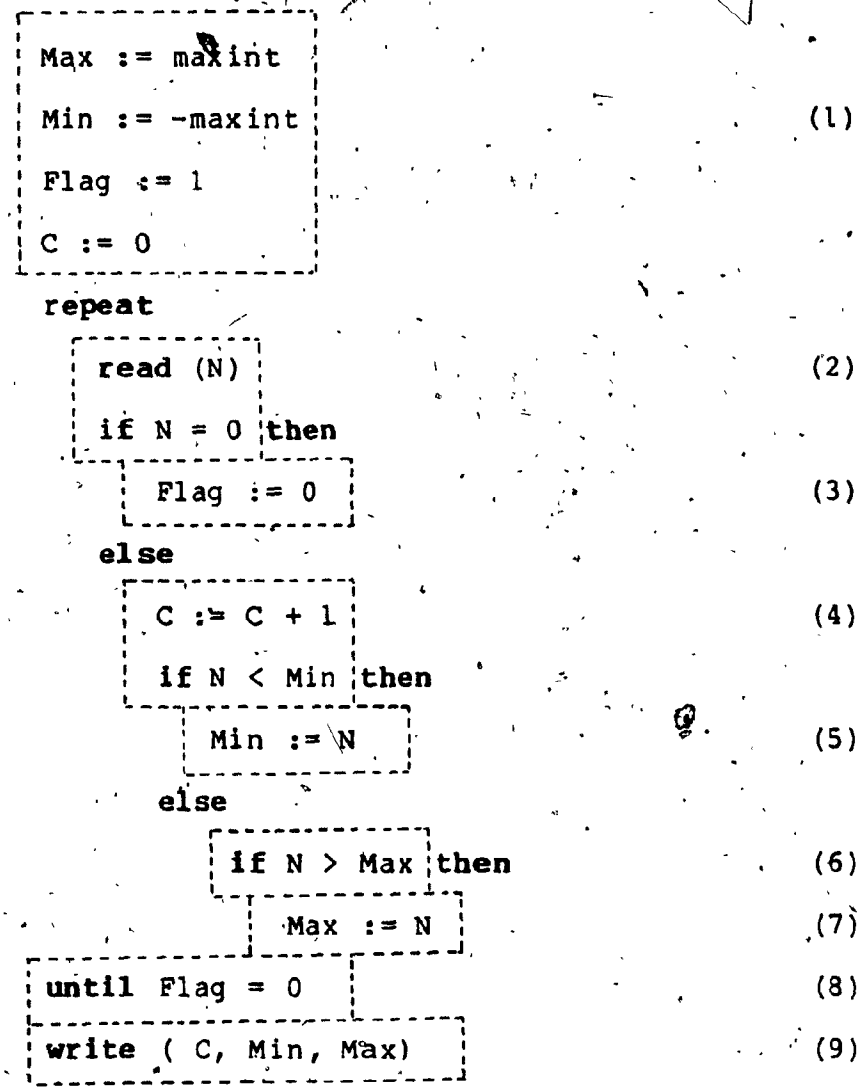


Figure 1.1 - Program and its basic blocks.

basic block may be constructed as follows:

a) The leaves are the statement operands, that is, the variable names and constants.

b) The interior nodes are labeled with the operators.

c) Interior nodes may be additionnally labeled by the computed values they represent.

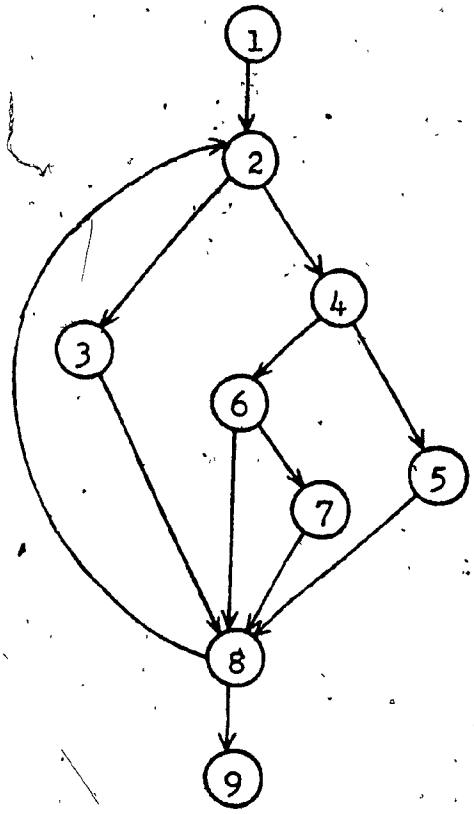


Figure 1.2 - Flow graph of the program of figure 1.1.

Data flow analysis local to a basic block is easily performed. During the construction of a DAG, common subexpressions, for example, can be automatically detected through the labels representing computed values attached to the interior nodes. These common subexpressions can be eliminated by considering only local data flow information. Using the notion of basic block we can define local and global optimization more precisely as follows: when the improvement transformations are obtained by considering only data flow at the interior of a basic block, we call it

'local optimization'. 'Global optimizations' require the analysis of the flow graph [Aho77b]. In the next section we enumerate the most typical data flow analysis problems.

1.4 Data Flow Analysis Problems.

A list of possible transformations to improve compiled programs was given in section 1.2. We will consider now the problem of how these transformations may be carried out. This must be done by a data flow analysis of the program. For example, to eliminate redundant subexpressions all such subexpressions must be found. Each type of transformation requires different data flow information and, therefore, different techniques are used to gather such information. In global optimizations, we find two classes of problems depending on when the information is required at a given point in the program flow graph: before control reaches that point (top of a node for example) or after control leaves that point (bottom of a node) [Kenn81]. In the first case the information is gathered by following the control represented by the flow graph from the root to the leaves and these problems are called "forward" flow problems. In the second case, the information is gathered backwards and the problems requiring such technique are usually called "backward" flow problems.

Gathering the information about data flow consists of finding the largest or smallest solution to a system of

simultaneous data flow equations. The system of data flow equations to be solved for specific problems is presented in the following chapters. We now describe some typical data flow analysis problems.

(i) Forward flow problems.

(a) Reaching definitions. We find out which definitions reach the top of a node in a flow graph. A variable or expression is said to be defined at a point **p** in the flow graph if it appears at the left side of an assignment, that is, if its value is changed. Information about reaching definitions is useful to several optimizations such as redundant subexpression elimination, constant propagation, and variable propagation.

(b) Available expressions. An expression A **op** B is available at a point p in a flow graph if every path that the program may take to p, evaluates A **op** B after the last definition of A or B. Finding available expressions is useful to redundant subexpression elimination.

(c) Copy propagation. This data flow analysis problem will look for assignments of the form A:=B. Together with reaching definitions, copy propagation will allow variable propagation optimization.

(ii) Backward flow problems.

(a) Live variables. We say that a variable A is live at a point p in the flow graph if the value of A at p is going to be used forward in some path starting in p. Live variables information allows further register allocation optimization.

(b) Very busy expressions. An expression A **op** B is very busy at point p in the flow graph if the same expression is evaluated along every path from p before any redefinition of A or B. Finding very busy expressions allows code hoisting transformations which consist in moving the busy expression to the point p, and having one computation of A **op** B only.

(iii) Data flow analysis problems for loop optimization.

(a) Loop invariant computations. Using reaching definitions information, this problem consists in marking those instructions which are invariant inside a loop. Loop invariant computations may be used to do code motion transformations.

(b) Detection of induction variables. Finding the induction variables in a loop allows induction variable elimination and strength reduction optimizations.

For example, to find available expressions at node x of

a flow graph, the intersection over the set of available expressions at the exit of every predecessor of x is calculated. Live variables, on the other hand, require set union operation over the set of live variables at the top of the successors of x. Thus, depending on the direction in which the information is propagated and on the operator used over the paths of the flow graph, we have four types of data flow analysis problems [Ullm75,Aho77b,Hech77] as ilustrated

	Set intersection	Set union
Forward flow problems	Available expressions	Reaching definitions
Backward flow problems	Very busy expressions	Live variables

Figure 1.3 - Types of data flow analysis problems.

by Figure 1.3.

1.5 Data Flow Analysis Methods.

The data flow analysis methods described in the literature have been classified in three main categories: iterative, interval analysis and global data flow analysis frameworks [Ullm75, Hech77]. A brief description of them follows.

(a) Iterative [Aho77b, Cock70, Grah76, Hech75, Kam76, Kenn71, Kenn75, Kenn76, Kou77, Shar78, Taj76, Ullm73]. This method consists in iterating through the nodes of the flow graph and applying the appropriate data flow equations until a fixed point is reached in the flow of information. Iterative methods are discussed in Chapter 2.

(b) Interval analysis [Alle76, Hech72, Hech74, Hech75, Kenn76, Tarj74]. A flow graph may be partitioned in regions called intervals. These intervals become the nodes of a new flow graph which may be partitioned again. Repeating this process, a derived sequence of flow graphs may be found with the last graph in the sequence eventually consisting of a single node. The interval analysis method uses these reduced control flow graphs to a fast gathering of data flow information. The propagation of information is a two-phase process. The first phase processes the derived sequence from the low order to the high order graph. The second phase reverses the process. This method is presented in Chater 3.

(c) Global Data Flow Analysis Frameworks [Fong75, Kam77, Kild73, Rose78]. This method was first considered by Kildall [Kild73] as an attempt to solve the data flow problems in an unified way. It is based in a finite semilattice framework. Some problems which cannot be represented by structures used in the two previous methods can be solved by this method. It is discussed in Chapter 4.

CHAPTER II

ITERATIVE METHOD

The iterative method is considered the simplest to implement and is also simple conceptually. This fact seems to make it popular among the practitioners. Several variations of the method have been discussed in the literature. Although we will discuss several versions of the iterative method in the last chapter for time-complexity comparison, in this chapter we will limite the discussion to the round-robin version applied to "available expressions" and "live variables" problems. A good exposition of the iterative algorithms may be found in [Hech77]. Other good references are [Aho77b] and [Kenn81].

The information or data to be propagated through the flow graph is usually represented by bit vectors (bit vector problems can also be modeled by a semilattice theoretic framework as discussed in Chapter 4). A bit vector is associated with each node of the flow graph. The bit positions represent variables or expressions of the program. The existence of an attribute for a variable or expression at a particular node will be indicated by setting up the corresponding bit of the associated bit vector. We will now focus our discussions on the available expressions problem.

2.1 Available Expressions.

We say that an expression A **op** B is "defined" at a given point if the expression is computed at that point. The expression A **op** B is "killed" if any or either of its operands is redefined. An expression A **op** B is then "available" at a point p of the flow graph G=(N,E,s) if every possible executable path leading to p contains a definition of such expression after the last definition of A or B.

Available expressions problem is a forward problem and its solution will provide information necessary to the elimination of redundant computations. To gather this information we will use a bit vector AVAIL(x) for each node x in N. This bit vector will contain a one at the position i if the expression i is available entering node x. We also need the vectors NKILL(x) and DEF(x) to keep local information to each node x. NKILL(x) will contain the set of expressions not killed in node x, and DEF(x) will contain the set of expressions defined in node x. The length of the vectors is m where m is the number of expressions.

Since we want a safe solution and we assume that every path may be possibly executed, the set of available expressions entering node x is obtained by the intersection of AVAIL(y) for every y in N which is a predecessor of x. This leads to the following system of equations to be solved:

$$AVAIL(x) = \bigcap_{y \text{ in } P(x)} (DEF(y) \cup (AVAIL(y) \cap NKILL(y)))$$

where P(x) is the set of predecessors of node x. AVAIL(y) $\cap$ NKILL(y) gives the set of available expressions preserved through node y to which set the new definitions in node y ,DEF(y), are added.

We present now an algorithm to compute available expressions. Because we want the largest possible solution, that is, to know the maximun number of available expressions at each node of the flow graph, the algorithm starts with the assumption that all the expressions are available at each node except entering the initial node where no expressions are available. The solution is obtained by iterating through the nodes of the flow graph and eliminating only those expressions found not available along some path.

ALGORITHM 2.1 - Computes available expressions.

Input: - A flow graph G=(N,E,s).
- Bit vectors DEF(i) and NKILL(i), 1<=i<=n.

Output: Bit vectors AVAIL(i), 1<=i<=n.

Method: Procedure AVAILEX.

```
Procedure AVAILEX
   AVAIL(1):= all 0's
   for i:=2 to n do AVAIL(i):= all 1's endfor
   CHANGE:= true
   while CHANGE do
       CHANGE:= false
       for j:= 2 to n (* in arbitrary order *)
          do
              previous:=AVAIL(j)
              AVAIL(j):= ⋂ (DEF(k) ⋃ (AVAIL(k) ⋂ NKILL(k)))
                         k∈P(j)
              if previous ≠ AVAIL(j) then
                  CHANGE:= true
              endif
       endfor
   endwhile
endAVAILEX
```

The algorithm stops when there is no change in a whole iteration through the nodes of the flow graph.

Algorithm 2.1 visits the nodes of the flow graph in arbitrary order. A more efficient algorithm can be found if the nodes are visited in reverse postorder (rpostorder). Rpostorder is a useful ordering of the nodes of the flow graph which is found by first, starting at the initial node, trying to visit nodes as far away from the initial node as quickly as possible (depth-first search) and , then,

reversing the order in which each node was last visited [Tarj72, Aho74, Hech75, Aho77b, Hech77]. The algorithm below, adapted from [Hech75], computes rpostorder. Trying to make paths as long as possible, the algorithm constructs and traverses a tree called "depth-first spanning tree" (DFST). Figure 2.1 gives one example of DFST and rpostorder.

ALGORITHM 2.2 Computes rpostorder for the nodes of a flow graph.

Input: A flow graph G=(N,E,s) represented by a successor lists (SUC). Initially, the nodes of G are numbered arbitrarily from 1 to n.

Output: A numbering of the nodes of G from 1 to n in reverse postorder (in array rPOSTORDER).

Method: - The output is stored in the global integer array rPOSTORDER[1:n].

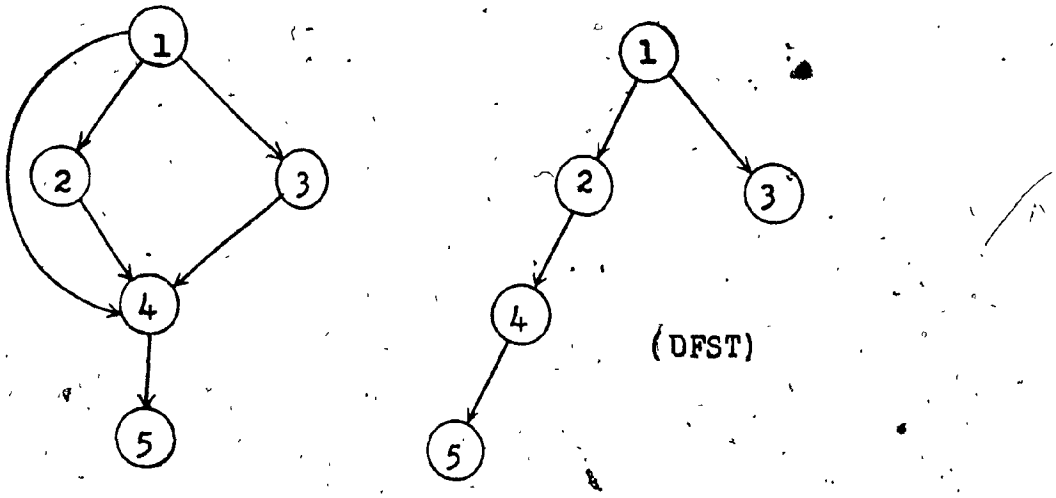
- Initialize a global integer variable i to n and all nodes of G to "new".

- Call DFS(s).

```
Recursive procedure DFS(x)
    mark x "old"
    While SUC(x) is not empty
        do
            select and delete a node y from SUC(x)
            if y is marked "new"
                then
                    call DFS(y)
                end
        end
    rPOSTORDER[x]:=i
    i:=i-1
endDFS
```

If (x,y) is an edge in a DFST, then x is a parent of y and y is a child of x, and rpostorder(y) > rpostorder(x). The rpostorder allows then to visit each node after all its predecessors have been visited in the DFST.

Using rpostorder we can obtain fast algorithms. The while statement of algorithm 2.1 for the computation of available expressions will have an upper bound equal to the number of nodes of the flow graph. Intuitively, we only need to consider cycle-free paths to compute available expressions; the information is propagated forward in the direction of the flow graph, and each node is visited after its predecessors and following the longest path whose upper



Preorder traversal: 1 2 4 5 4 2 1 3 1

Last ocurrency : 5 4 2 3 1

Reverse postorder : 1 3 2 4 5

Figure 2.1 - DFST and Rpostorder.

bound is the number of nodes of the flow graph.

Another property of rpostorder is that it helps to discover an important parameter of flow graphs called "depth". The depth of a flow graph is defined to be the largest number of backward edges on any cycle-free path. When constructing a depth-first spanning tree, an edge (x,y) of the flow graph is a backward edge if and only if rpostorder(x)>= rpostorder(y). Intuitively the depth of a flow graph is an indication of the loop nesting in the graph and it is usually small (about 3 in the average). The number of passes through the while-loop of the algorithm 2.1 will be limited to two more than the depth of the flow graph

as defined above [Aho77b]. We present now an improved version of the algorithm 2.1 using rpostorder.

ALGORITHM 2.3 - Computes available expressions.

Input: - A flow graph G=(N,E,s) with nodes numbered from 1 to n by rpostorder. Each node is refered by its rpostorder number.

- Bit vectors DEF(i) and NKILL(i), 1<=i<=n.

Output: Bit vectors AVAIL(i), 1<=i<=n.

Method: Procedure AVAILEX.

```
Procedure AVAILEX
   AVAIL(1):= all 0's
   for i:=2 to n do AVAIL(i):= all 1's endfor
   CHANGE:= true
   while CHANGE do
      CHANGE:= false
      for j:= 2 to n do (* rpostorder *)
         previous:=AVAIL(j)
         AVAIL(j):= ⋂ (DEF(k) ⋃ (AVAIL(k) ⋂ NKILL(k)))
                    k∈P(j)
         if previous ≠ AVAIL(j) then
            CHANGE:= true
         endif
      endfor
   endwhile
endAVAILEX
```

The algorithm terminates when the propagation of information stabilizes.

2.2 Live Variables.

Live variables problem is a typical backward data flow analysis problem. It is necessary to determine if a variable is going to be used (needed) later on in the program before being redefined. A solution of this problem allows an improved allocation of registers and helps in loop optimization. Formally, we say that a variable x is 'live' at a point p in the flow graph G=(N,E,s) if there is any path from p to a use of x without redefinition of x after p.

As for available expressions, live variables information may be represented by bit vectors. Let LVTOP(x) be the bit vector which contains the set of live variables on entry to node x, and let LVBOT(x) be the set of variables which are live on exit of node x. To represent local information, let USE(x) be the set of variables used in x before any possible definition in x and let THRU(x) be the set of variables not defined in x. We have, then, the following system of equations to be solved:

$$LVBOT(x) = \bigcup_{y \in S(x)} ((LVBOT(y) \bigcap THRU(y)) \bigcup USE(y)$$

where S(x) is the set of successors of x. The iterative algorithm presented below visits the nodes of the flow graph

in rpostorder. It has been adapted from [Hech75].

ALGORITHM 2.4 - Computes live variables.

Input: - A flow graph G = (N,E,s). The nodes are numbered from 1 to n by rPOSTORDER.

- Sets USE(i) and THRU(i), 1<=i<=n represented by bit vectors of length m where m is the number of variables.

Output: Bit vectors LVBOT(i), 1<=i<=n.

```
Procedure LIVEVAR
   for j:=1 to n do LVBOT(j):= all 0's end
   CHANGE:= true
   while CHANGE do
      CHANGE := false
      for j := n to 1 by -1 do
          previous := LVBOT(j)
          LVBOT(j):= ⋃ ((LVBOT(k) ⋂ THRU(k)) ⋃ USE(k))
                     k∈S(j)
          if previous ≠ LVBOT(j) then CHANGE := true end
      endfor
   endwhile
endLIVEVAR
```

Algorithm 2.4 terminates if no LVBOT(j) is changed in the iteration. The efficiency of iterative algorithms will be discussed in Chapter 6.

CHAPTER III

INTERVAL ANALYSIS

3.1 Reducibility by Intervals.

Interval analysis is another approach to solve global data flow analysis problems. There is a class of flow graphs called 'reducible' flow graphs for which data flow analysis is easily performed. These flow graphs can be partitioned into regions called 'intervals'. Intuitively, intervals represent loops of the flow graph. After partitionning a flow graph into intervals, we can construct another flow graph whose nodes are the already found intervals, and there is an edge from interval I to interval J if there is an edge from a node in interval I to a node in interval J. By repeating this process we can obtain a sequence of flow graphs whose nodes are basic blocks in the first flow graph, innermost 'loops' (intervals) in the second flow graph in the sequence and outer 'loops' are found by partitionning the following flow graphs in the sequence. This way we can find a nested structure of loops and global data flow analysis can be performed efficiently. Local data flow information is propagated from innermost 'loops' to outermost 'loops' and then the process is

reversed.

Formally, in a flow graph G = (N,E,s), an interval I with header h, denoted I(h), is constructed as follows:

1. I(h) := {h}

2. While there is a node m <> s and m $\notin$ I(h) and all predecessors of m are in I(h), do I(h) := I(h) $\cup$ {m}.

Once a first partition of G has be found, the derived flow graph I(G) can be constructed as follows:

(i) The nodes of I(G) are the intervals of G.

(ii) There is an edge from interval I to interval J if there is an edge from a node in I to the header of J and I $\neq$ J.

(iii) The initial node of I(G) is I(s).

G is called the 'first order graph' and I(G) the 'second order graph'. I(G) may now be partitioned into 'second order intervals' and so on to obtain a 'derived sequence' of G. Formally, a sequence G = G1, G2,...,Gk is called the derived sequence of G iff Gi+1 = I(Gi) for 0<=i<k, Gk-1 $\neq$ Gk, and I(Gk) = Gk. Gk is called the 'limit flow graph' of G [Hech74].

A graph G is said to be 'reducible' if and only if the limit flow graph Gk consists of a single node. In such a case Gk is called the 'trivial' flow graph. The flow graph G is said to be 'irreducible' if the graph G does not have a limit flow graph consisting of a single node.

Reducibility has many other interesting properties. It allows to find the loops of a flow graph unambiguosly. These loops are defined by the backward edges of the flow graph. In fact a flow graph G is reducible if and only if the edges of the flow graph can be partioned in two classes: forward edges which form an acyclic graph whose nodes can be reached from the initial node of G, and backward edges whose heads 'dominate' their tails [aho77] (we say that a node x dominates a node y if every path from the initial node of the flow graph to y passes through x).

If G=(N,E,s) is a flow graph and I(h) is an interval of G then,

a) Every edge entering a node of I(h) from the outside enters the header h, that is, an interval has a single entry.

b) The header h dominates every other node in I(h).

c) Every cycle in an interval I(h) includes the header h.

d) The interval I(h) is unique for each h in G and independent of the order in which candidates for m are chosen in the definition of interval.

Here we give an algorithm to partition a flow graph into intervals. It was taken from [Alle76]. Figure 3.1 ilustrates the derived sequence of a reducible flow graph.

ALGORITHM 3.1 - Interval partition.

Input: A flow graph G=(N,E,s).

Output: A set of disjoint intervals L.

Method:

```
H:={s} (* H is the set of potential header nodes *)
  while H not empty do
     select and delete a node h from H
     (* find I(h) from the definition of interval *)
     I(h) := {h}
     while there is an x in (S(I(h)) - I(h))
     such that P(x) ⊂ I(h) do
       I(h) := I(h) ⋃ {x}
     endwhile
     L := L ⋃ {I(h)}
     H := H ⋃ (S(I(H)) - I(h))
  endwhile
```

The order in which nodes are added to an interval is called 'interval order'. If the nodes of an interval I are processed in 'interval order' then a node x in (I - {h}) will be processed only after every predecessor of x has been processed. This order is important in data flow analysis.

The existence of irreducible flow graphs seems not to be a major problem to the application of the interval method to solve data flow analysis problems. Fortunately, most of the flow graphs for computer programs appear to be reducible in practice. Knuth [Knut71] found that in a sample of 50

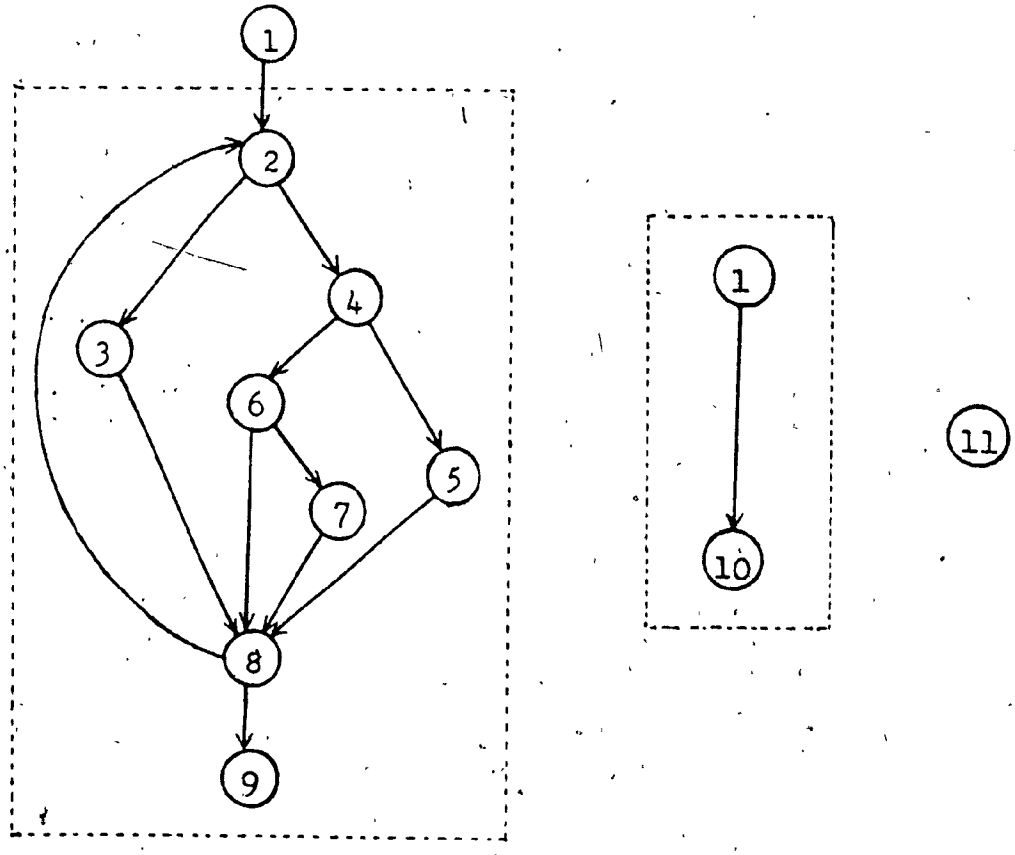


Figure 3.1 - Derived sequence of a reducible flow graph.

FORTRAN programs none was irreducible. With the use of structured programming techniques, reducible programs are normally obtained.

Irreducible flow graphs are characterized, intuitively, by having a loop with two entries. This is the case ilustrated by the 'paradigm irreducible flow graph' of figure 3.2 (a). Neither node 2 nor node 3 dominates each other. Irreducible flow graphs can be made reducible by 'node splitting'. Node splitting is a technique which consists of making as many identical copies of a node as

there are edges entering such a node (and no self-loop). Figure 3.2 (b) ilustrates an equivalent reducible flow graph of the irreducible one in (a) after node splitting.

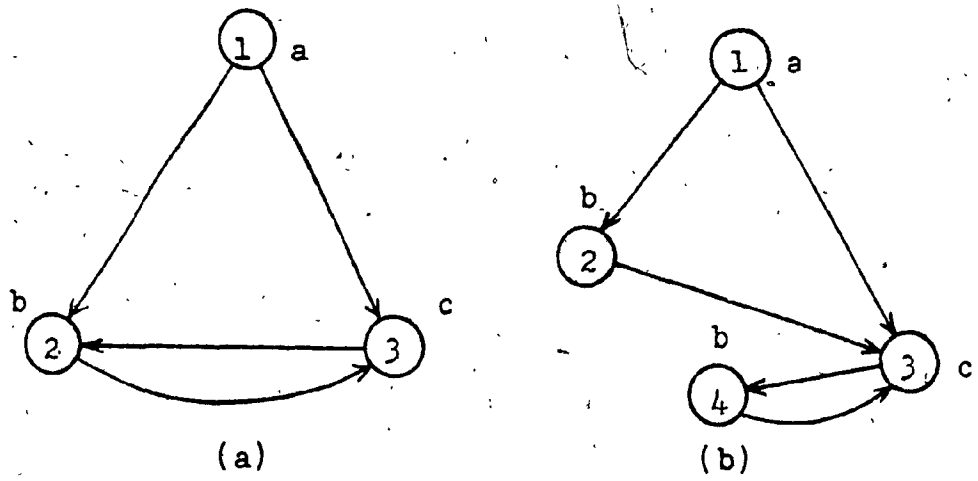


Figure 3.2 - (a) The paradigm irreducible flow graph, (b) reducible after splitting.

Node splitting is used for analysis of data flow and it does not implies actual duplication of code. In section 3.3 we will discuss how the reaching definitions algorithm of Allen and Cocke [Alle76] handles irreducible flow graphs by node splitting. Another way of dealing with irreducible flow graphs is by applying the iterative techniques presented in the previous chapter to obtain data flow information from irreducible subgraphs.

We finish this section by mentionning that Hecht and

Ullman [Hech72] introduced two graph transformations known as T1 and T2 and found that the class of reducible flow graphs by intervals is exactly the class of reducible flow graphs by the transformations T1 and T2.

Transformation T1 is the removal of a self-loop, that is, the removal of an edge from a node to it self. Transformation T2 replaces two nodes x and y of the flow graph by a new node z if x is the only predecessor of y and y is not the initial node. The application of these two transformations to a flow graph until no longer possible, and independent of the sequence, will result in a unique flow graph. If this unique resulting flow graph is the trivial flow graph then the flow graph is reducible. Reducibility by T1 and T2 is called 'collapsibility'.

3.2 Data Flow Analysis Using Intervals.

Data flow analysis based on intervals usually requires two-pass algorithms. In fact, to propagate the information about data flow properly, we need information about predecessors (successors) as well as local information of each node. The nodes are basic blocks in the first order graph, but they are intervals in second order graphs and higher order graphs. During the first pass, local information about each node is collected. Once the information is obtained for each basic block, the algorithm collects information about first order intervals by

processing the nodes in each interval in the interval order. The information is then posted to the node representing each interval in the second order derived graph. The process is then repeated until 'local' information is available for the trivial flow graph.

During the second pass, the graphs of the derived sequence are processed from high order to low order.

Since the nodes of a derived flow graph are intervals, there are different paths through the interval from its header to possibly different exits. Thus the information at each exit may not be the same. The information is, then, associated with the edges which leave each node and not with the node as was the case in the iterative method. For this purpose, an edge is assumed leaving each leaf of the flow graph.

An excellent discussion of the interval method applied to the problem of reaching definitions is found in [Alle76]. To ilustrate the method, we will present in the next sections an algorithm to solve reaching definitions from [Al176] and another to solve live variables problem taken from [Kenn81].

3.3 Reaching Definitions Problem using Intervals.

Reaching definitions is a forward data flow analysis problem. We want to know which definitions reach the top of each basic block of the flow graph. Let denote by Ri the

set of definitions which reach node ni. Let DBj be the set of variable definitions from the tail node of edge j and let PBj the set of definitions preserved through the tail node of edge j. We obtain the set of available definitions on edge j, Aj, by the formula

$$Aj = (Rj \cap PBj) \cup DBj$$

for each edge j with tail node i and

$$Ri = \bigcup_{p} Ap$$

for each entering arc p to i, i<>s.

We reproduce now the algorithm of Cocke and Allen, in its english version, for solving those equations.

ALGORITHM 3.2 - Reaching definitions using intervals.

Inputs

1. The ordered set of graphs (G1,G2,...,Gn) determined by interval analysis.
2. The intervals in each graph with their nodes given in interval order.
3. Definitions defined and preserved on each edge in the first order graph (DB and PB sets).

Outputs

1. A set R of the definitions that reach each node.
2. A set A of the definitions available on each edge.

Steps

Phase I

1. For each graph, Gg, in the order G1,G2,...,Gn-1, perform steps 2 and 3.

2. If the current graph is not G1 then initialize the PB and DB for the edges of the graph. This is done by first identifying the edge in Gg-1 to which each edge in Gg corresponds (these will be interval exit edges). Then using the information generated during step 3 for Gg-1, for each edge i in Gg with corresponding exit edge x from interval with head h in Gg-1, set:

2.1. PBi = Px and

2.2. DBi = (Rh $\cap$ Px) $\cup$ Dx

3. For each exit edge of each interval in Gg determine P, the definitions preserved on some path through the interval to the exit, and D, the definitions in the interval that may be available on the exit. These are determined by finding P and D for each edge in the interval:

3.1. For each exit edge i of the header node:

Pi = PBi

Di = DBi

3.2. For each exit edge i of each node j (j=2, 3,....) in interval order:

Pi = ($\bigcup_p$ Pp) $\cap$ PBi

Di = (($\bigcup_p$ Dp) $\cap$ PBi) $\cup$ DBi for all p input edges to node j.

While processing an interval determine the set of definitions, Rh, that can reach the interval head, h, from the inside the interval by:

$Rh = \bigcup_l Dl$

for all interval edges l which enter h,(latching edges). If there are none set Rh = empty set.

Between phase I and II the R vector for the single node in the nth order derived graph is initiated to the set of definitions known to reach the program from outside.

Phase II

1. For each graph, Gg, in the order Gn-1,...,G2, G1, perform steps 2 and 3.

2. For each node i in Gg+1 form $Rh = Rh \bigcup Ri$ where h is the head of the interval in Gg which i represents in Gg+1.

3. For each interval process the nodes in interval order determining the definitions reaching each node and available on each node exit edge as follows:

3.1. For each exit edge i of the header node h

$$Ai = (Rh \bigcap PBi) \bigcup DBi$$

3.2. For each node j (j=2, 3,...) in interval order first form

$$Rj = \bigcup_p Ap \quad \text{for all input edges p to j}$$

then for each exit edge i of j form

$$Ai = (Rj \bigcap PBi) \bigcup DBi$$

End of algorithm

The existence of irreducible flow graphs is transparent to the above algorithm. The nodes of a irreducible subgraph are treated as interval heads (each node is a unique interval) and the Phase I of the algorithm derive Pi and Di for each edge. The splitted form of this irreducible subgraph is treated as a higher order graph in the derived sequence. In Step 2 of Phase I the algorithm will initialize DBi and PBi for the edges in the splitted higher order graph using P and D values as necessary, that is, if

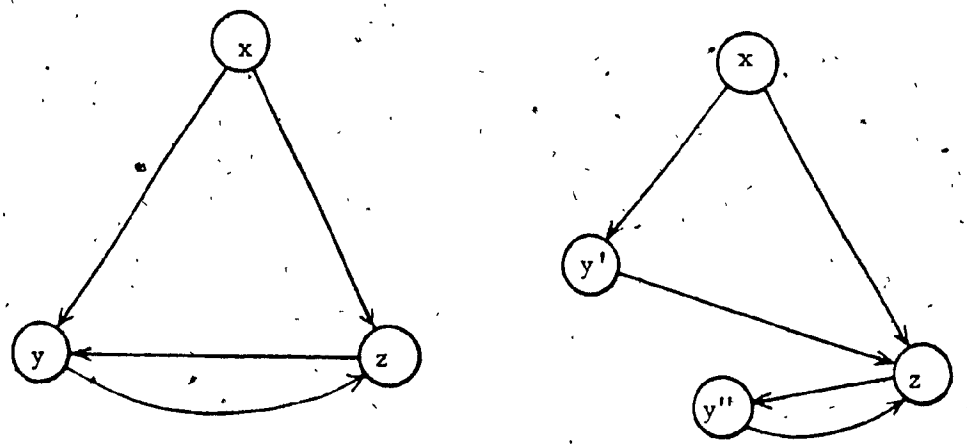


Figure 3.3 - Irreducible subgraph and its equivalent reducible.

node y has been splitted in y' and y" (see figure 3.3) and there was an edge (y,z) in the irreducible flow graph, the values P and D of th edge (y,z) will be picked twice, one for the edge (y',z) and other for the edge (y",z) of the

splitted flow graph. During phase II, step 2 will find reaching definitions at the top of node y, that is, Ry=Ry $\cup$ Ry' and then Ry=Ry $\cup$ Ry" giving the wanted result Ry=Ry' $\cup$ Ry" (Ry being initially null).

3.4 Live Variables Problem using Intervals

The live variables problem was already defined in the previous chapter. Since live variables is a backward problem, the propagation of data flow information is done from the leaves of the flow graph to the root. The information about live variables is once again represented by bit vectors.

Let LVTOP(x) be the set of live variables on entry to node x. Let THRU(x,y) be the set of variables not defined (not changed) in x on the path from x to y where y is a successor of x. THRU(x,y) represents local information which is in this case associated with the edges. We need also the set of variables used in x before any possible definition in x. Let USE(x) contain such information. The system of equations to be solved may now be stated as follows:

$$LVTOP(x) = USE(x) \cup \bigcup_{y \in S(x)} (THRU(x,y) \cap LVTOP(y))$$

where S(x) is the set of successors of x. We give now the algorithms to solve the equation system.

3.4.1 An Interval Analysis Algorithm to Compute Live Variables.

(a) Pass 1. During the first pass, local information USE and THRU is computed for each node. Starting with the first order flow graph G, USE and THRU are computed for each basic block. Then the algorithm is applied to the nodes (intervals) of the second order flow graph and so on to the other flow graphs in the sequence until the quantities have been computed for the trivial flow graph Gk.

ALGORITHM 3.3 - Pass 1

Input: - An interval I
- USE(x), for all x in I.
- THRU(x,y), for all x in I, for all y in S(x).

Output: - USE(I)
- THRU(I,J), for all J in S(I).

Auxiliary: For each x in I, PATH(x) contains the set of variables which are not defined (there is a clear path) from the entry of I to the entry of x.

Method:

```
begin
  USE(I):= USE(h)  (* h is the header of I *)
  PATH(h):= set of all variables.
  for all x in (I - {h}) in interval order
    do
      PATH(x) := ⋃ (PATH(y) ⋂ THRU(y,x))
                 y ∈ P(x)
      USE(I) := USE(I) ⋃ (PATH(x) ⋂ USE(x))
    od
    (* let z denote the header of J *)
    for J such that z is in S(I)
      do
        THRU(I,J) := ⋃ (PATH(y) ⋂ THRU(y,z))
                     y ∈ P(z) ⋂ I
      od
end
```

(b) Pass 2. The second pass propagates data flow information from the derived flow graph of highest order (trivial flow graph) to the first order flow graph. LVTOP is actually computed using the information obtained during the first pass. The algorithm that follows processes the nodes of each interval in reverse interval order to solve the system of equations given above in Section 3.3. This order allows the processing of each node after all its successors have been processed.

ALGORITHM 3.3 - Pass 2.

Input: - An interval I with header h.

- USE(x), for all x in I.
- THRU(x,y), for all x in I, for all y in S(x).
- LVTOP(I)
- LVTOP(J), for all J in S(I).

Output: LVTOP(x), for all x in I.

Method:

```
begin
  LVTOP(h) := LVTOP(I)
   for all J in S(I)
     do
       LVTOP(header of J):= LVTOP(J)
     od
   for all x in (I - {h}) in reverse interval order
     do
       LVTOP(x) := USE(x) ∪ ⋃ (THRU(x,y) ∩ LVTOP(y))
                            y ∈ S(x)
     od
end
```

The proof of termination and correctness of these algorithms is given in [Kenn75].

CHAPTER IV

GLOBAL DATA FLOW ANALYSIS FRAMEWORKS

An attempt to find a general approach to data flow analysis was first considered by Kildall [Kild73]. In his paper Kildall discusses the application of global data flow frameworks to constant propagation and common subexpression elimination problems. This method presents the possibility of solving data flow analysis problems whose data cannot be represented by bit vectors. One example of these problems is type determination [Tene74]. The bit-propagation problems solved by the methods discussed in the previous chapters can be also modeled by this method. The data flow framework was stated by Kildall using a finite-meet semilattice and since then the same approach has been considered by several other authors [Fong75, Kam77, Hech77, Rose78, Kam76, Tene74, Shar81, Rose81]. This general approach to the solution of data flow analysis problems constitutes a unified method and should be considered as foundation for data flow analysis. Before going further, we will give some notions about lattice theory.

4.1 Some Theoretical Notions About Lattices.

To understand global data flow analysis frameworks we need to know some basic concepts of a branch of mathematics called 'lattice theory'. Most of the definitions and properties given in this section have been taken from [Hech77]. A good mathematical reference is [Grat71].

Let S be a set. A partial ordering on S is a relation on S denoted by $\leq$, which is:

i) Reflexive : for every a in S, $a \leq a$

ii) Anti-symetric : if $a \leq b$ and $b \leq a$, then $a = b$.

iii) Transitive : if $a \leq b$ and $b \leq c$, then $a \leq c$

A set S with a partial ordering $\leq$, denoted by $(S, \leq)$ is called a partial ordered set (or poset for short).

If $a \leq b$ we say that a is included in b. If $a \leq b$ and $a \neq b$ we may write $a < b$. $b \geq a$ means $a \leq b$ and $b > a$ means $a < b$. For example, if S is a collection of sets A, B,..., $A \leq B$ means that A is a subset of B; $A < B$ means that A is a proper subset of B.

If $(S, \leq)$ is a poset, then so is $(S, \geq)$. The poset $(S, \geq)$ is called the dual of $(S, \leq)$.

If $(S, \leq)$ is a poset, then $<$ is irreflexive and transitive. Conversely, if $<$ is an irreflexive and transitive relation on S and $x \leq y$ is defined by $x<y$ or $x=y$, then $(S, \leq)$ is a poset.

We say that a poset $(S, \leq)$ has a **zero element** if there is an element **0** in S such that $0 \leq x$ for all x in S. Similarly, we say that the poset has a **one element** if there exist an element 1 in S such that $x \leq 1$ for all x in S.

If $a \leq b$ or $b \leq a$ we say that a,b are **comparable** and **incomparable** otherwise. A poset $(S, \leq)$ in which there are no incomparable elements is called a **chain** (also called linear ordering). A chain on $S = \{s1, s2, \ldots, sk\}$ may be denoted by $s1<s2<s3<\ldots<sk$ and we say that it is of length k.

A <u>joint</u> (least upper bound) of a and b in the poset $(S, \leq)$ is an element c in S such that $a \leq c$ and $b \leq c$ and there is no x in S such that $a \leq x \prec c$ and $b \leq x \prec c$. A <u>meet</u> (greatest lower bound) of a and b is an element d in S such that $d \leq a$ and $d \leq b$ and there is no x in S such that $d \prec x \leq a$ and $d \prec x \leq b$. If the elements a and b of S have a unique joint, it is denoted by $a \vee b$ and if they have a unique meet, it is denoted by $a \wedge b$.

A <u>lattice</u> is a poset $(S, \leq)$ such that any two elements a, b in S have a unique joint and meet. A lattice is usually denoted by the triple $(S, \vee, \wedge)$ which is an algebra, that is, a set equipped with operations (in this case two binary operations). The joint and meet operations of a lattice $(S, \vee, \wedge)$ have the following properties :

For all x, y, z in S

(a) $x \wedge x = x$, $x \vee x = x$ Idempotency

(b) $x \vee y = y \vee x$, $x \wedge y = y \wedge x$ Commutativity

(c) $x \vee (y \vee z) = (x \vee y) \vee z$, Associativity

$x \wedge (y \wedge z) = (x \wedge y) \wedge z$

(d) $x \vee (x \wedge y) = x$, $x \wedge (x \vee y) = x$ Absorption

(e) $x \vee y = y$, $x \wedge y = x$, $x \leq y$ are equivalent.

A set S and two binary operations $\vee$ and $\wedge$ on S is a lattice if both $\vee$ and $\wedge$ are idempotent, commutative and associative.

We also say that a lattice $(S, \vee, \wedge)$ is

i) <u>bounded</u> iff it has both a **0** and **1** element.

ii) of <u>finite length</u> iff each chain in the lattice is finite.

iii) <u>distributive</u> iff for all x, y, z in S we have

$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ and

$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

A <u>semilattice</u> is a pair $(S,*)$, where S is a nonempty set and $*$ is a binary operation on S with the properties idempotent, commutative, and associative. A poset $(S, \leq)$ is a joint-semilattice, denoted by $(S, \vee)$, if any two elements a, b in S have a unique joint. Similarly, a poset $(S, \leq)$ is a meet-semilattice if any two elements a, b in S have a unique meet. If $(S, \vee, \wedge)$ is a lattice, then, $(S, \vee)$ and $(S, \wedge)$ are both semilattices.

We also say that a semilattice $(S,*)$,

i) has a zero element 0 if for all x in S, $0 * x = 0$.

ii) has a one element 1 if for all x in S, $x * 1 = 1$.

iii) is bounded if each chain in the semilattice is of finite length.

Let $(S, \leq)$ be a poset. An operation f: S ---> S on S is called <u>monotonic</u> iff for all x,y in S, $x \leq y$ implies $f(x) \leq f(y)$.

If f and g are monotonic operations on a poset, then so is fg. Monotonicity is preserved by composition.

A sequence x1, x2, ... in a poset $(S, \leq)$ of finite length converges to an element t in S if and only if there exist a k>=0 such that for all i>=k we have xi = t. We can state now a **fixed point** theoreme.

Let f: S ---> S be a monotonic operation on a poset $(S, \leq)$ with a zero element 0 and finite length. The <u>least fixed point</u> of f is $f^k(0)$, where $f^0(x) = x$, $f^{i+1}(x) = f(f^i(x))$ for $i>=0$, $f^k(0) = f(f^k(0))$ and there is no j, $0<=j<k$ such that $f^j(0) = f(f^j(0))$.

If $(S, \leq)$ is a semilattice, an operation f: S--->S on S is called,

a) <u>Distributive</u> (meet-endomorphism) if and only if for all x, y in S $[f(x \wedge y) = f(x) \wedge f(y)]$.

b) <u>Monotonic</u> if and only if for all x, y in S, $[f(x \wedge y) \leq f(x) \wedge f(y)]$.

The expression [$f(x \wedge y) \leq f(x) \wedge f(y)$] of condition b) is what Hecht [Hech77] calls the GKUW property (for the authors Graham, Kam, Ullman, and Wegman which observed it) and it is equivalent to for all x,y in L, [$x \leq y$ implies $f(x) \leq f(y)$] which defines the monoticity of a poset as previously seen.

4.2 Monotone Data Flow Analysis Framework.

A global data flow framework is defined to be a triplet $(L, \wedge, F)$, where $(L, \wedge)$ is a meet semilattice of data flow information with the meet operation $\wedge$. F is a space of functions acting on L. The data flow information to be propagated through a flow graph constitutes the elements of L. The way in which such information is propagated along the flow graph paths is described by F. Global data flow frameworks modeled by a semilattice having the distributive property are called distributive frameworks and they have been studied by Kildall [Kild73]. But some data flow problems are not distributive [Kam75]. A more general approach have been studied by Kam and Ullman [Kam75] by considering monotone frameworks. Monotonicity is a weaker condition than distributivity and monotone frameworks seem to model the known data flow analysis problems. The definition of monotone frameworks that follows comes from [Kam75]

A set of functions F acting on a semilattice L of finite length with zero element, is said to be a "monotone function space associated with L" if the following conditions are satisfied:

(a) Each f in F satisfies the monotonicity condition, that is, for all x,y in L, and for all f in F, $[f(x \wedge y) \leq f(x) \wedge f(y)]$

(b) There exists an identity function e in F, such that for all x in L, [e(x) = x]

(c) F is closed under composition. That is, f,g in F implies fg in F, where for all x,y in L, [fg(x) = f(g(x))].

(d) L is equal to the closure of 0 under the meet operation and application of functions in F. That is, for each x in L, there exists an f in F such that x = f(0).

A monotone data flow analysis framework is a triplet $(L, \wedge, F)$, where

(1) $(L, \wedge)$ is a bounded semilattice with meet $\wedge$.

(2) F is a monotone function space associated with L.

An "instance" of a monotone framework is a pair I=(G,M), where

(1) G = (N,E,s) is a flow graph.

(2) M: N ---> F is a function which maps each node in N to a function in F.

As an example of a monotone data flow analysis

framework we consider constant propagation which is a typical problem discussed in the literature [Kild73, Kam75, Hech77]. The monotone data flow framework which models the problem is a triplet $(L, \wedge, F)$. Here L is a set of functions from finite subsets of V to R, where $V = \{A1, A2, A3, \ldots\}$ is an infinite set of variables and R is the set of all real numbers. Therefore the elements of L may be viewed as finite subsets of VxR. Intuitively, x in L represents the information about variables which we may assume at certain points of the flow graph. The tuplet (A,r) in x means that variable A has value r. The meet operation $\wedge$ in L is set intersection. L has a **zero** element which is the empty set. F is the function space acting on L. Informally, the data flow information to be propagated may be represented by an ordered set of tuplets (A,r), where $A \in V$ and $r \in R$. Such an ordered set is then associated with each node of the flow graph. These ordered sets are the elements of L. The meet operation over the ordered sets propagates the information along the paths of the flow graph. The functions in F acting on L may be seem as the effect of the nodes in the ordered sets of tuplets, that is, if node n contains the statement A:=r, then we denote its effect by the function <A:=r> which is applied to the semilattice element x associated with n. By simplicity we assume that the nodes of the flow graph contain only assignment statements of the form A:=r and A:=B **op** C. Then there are functions denoted <A:=r> and <A:=B **op** C> in F, for

each A, B and C in V, r in R and **op** in {+,-,*,/}. If x is in L then <a:=r>(x)=y, where the new set of tuplets y = x $\bigcup$ {(A,r)} - (any other tuplet in x with A in the left side). If the statement A:=B **op** C is found, then we will have <A:=B **op** C>(x) = y, where, if (B,b) and (C,c) are in x, then we do as in the previous case but using the value of 'b **op** c' for r. If (B,b) or (C,c) are not in x then B **op** C has not a constant value and (A,r) is undefined in y. Each node of the flow graph may be modeled by the function composition of the functions representing statements within it.

A monotone framework $(L, \bigwedge, F)$ is said to be distributive if and only if

$$(\forall f \in F)(\forall x,y \in L)\ [f(x \bigwedge y) = f(x) \bigwedge f(y)]$$

Distributive frameworks constitute a subclass of monotone frameworks. The maximum fixed point (MFP) solution to distributive frameworks coincides with the 'meet over all paths' solution (MOP solution). Intuitively, the MOP solution is the calculation of the maximum relevant data flow information for each node of the flow graph. This information is obtained considering every possible execution path from the initial node to each node of the flow graph.

Kam and Ullman [Kam75] showed that constant propagation is not distributive. To show that it is not distributive, a counter example may be used. Considering the flow graph of figure 4.1, we have that <C:=A+B>(x $\bigwedge$ y) = $\varnothing$ and

$\langle C:=A+B\rangle(x) \wedge \langle C:=A+B\rangle(y) = \{(C,3)\}$ then

$\langle C:=A+B\rangle(x \wedge y) \neq \langle C:=A+B\rangle(x) \wedge \langle C:=A+B\rangle(y)$

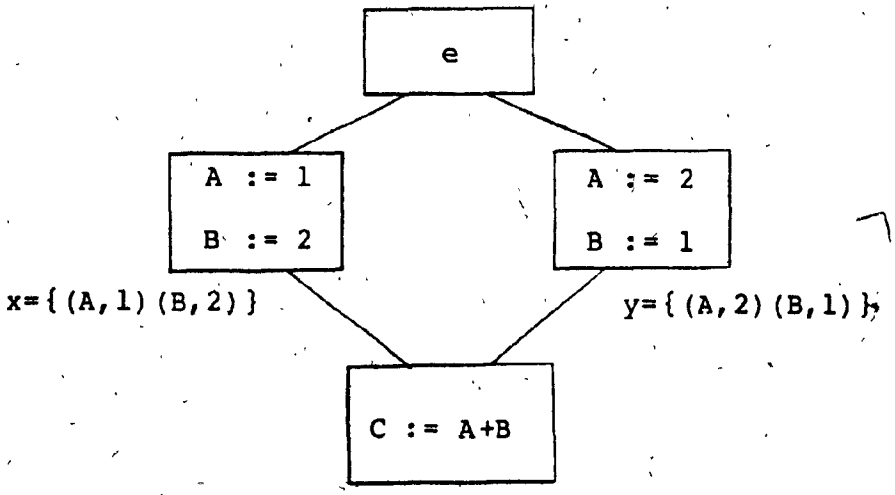


Figure 4.1 - Distributivity counter example.

As an example of distributive framework we can consider again the available expressions bit-vector problem discussed in previous chapters. Let AVAIL = $(L, \wedge, F)$ be the framework which models the problem, where

(i) L is the set of 2^m m-bit vectors (m = number of expressions).

(ii) AND is the meet operation $\wedge$.

(iii) F is the function space associated with L and will consist of kill and generate operations.

If a node x has k and g bit vectors for kill and generate respectively, then the function can be represented

by <k,g> and for all x in L, [<k,g>(x) = (x AND NOT k) OR g]. The zero element of L is the vector of all 0's, and the one element is the vector of all 1's. It is easy to show that for all x,y in L, and for all f in F, [f(x $\bigwedge$ y) = f(x) $\bigwedge$ f(y)].

4.3 The General Iterative Algorithm.

In the previous section we have illustrated with two examples how the data flow information is propagated by the action of some functions on the elements of a semilattice L and its meet operation. We can now present the system of equations to be solved.

Let I=(G,M) be an instance of a monotone framework (L, $\bigwedge$,F), where G = (N,E,s) is a flow graph whose nodes are numbered from 1 to n by rpostorder. The maximum fixed point solution of I is the maximum fixed point solution of the following system of equations [Kild73,Kam75,Hech77]

$$X[1] = \mathbf{0}$$

$$X[i] = \bigwedge_{j \in P(i)} f_j(X[j]) \quad \text{for } 2 \leq i \leq n$$

where P(i) is the set of predecessors of i and f_j is the operation associated with node j.

Kildall's general iterative algorithm converges to the MFP solution of the framework, independent of the order in which the nodes of the flow graph are visited. We give below the round-robin version of Kildall's algorithm in

which the nodes are visited in rpostorder. The algorithm has been adapted from [Kam76].

ALGORITHM 4.1 - Forward analysis iterative algorithm.

Input : An instance I=(G,M) of a monotone framework $D=(L, \bigwedge, F)$, where G=(N,E,s) is a flow graph with the nodes numbered from 1 to n by rpostorder.

Output: An array A[1:n] of semillattice elements.

Method: - A[1:n] is a global array and A[j] is associated with node j.

- Procedure GIA below.

```
Procedure GIA
semilattice element TEMP
integer j
boolean CHANGE
(* initialize *)
A[1] := 0
for j:= 2 to n by 1 do
   A[j]:= ⋀ f_g(A[g])
          g ∈ P*(j)
endfor
(* iterate *)
CHANGE:= true
while CHANGE do
    CHANGE:= false
    for j:=2 to n by 1 do
```

```
        TEMP:= ⋀ f_q(A[q])
               q ∈ P(j)
        if TEMP ≠ A[j] then
            CHANGE:= true
            A[j]:= TEMP
        endif
    endfor
endwhile

return
```

The set of predecessors of node j, P*(j), is defined as follows

P*(j) = {q | q in P(j) and q<j in rpostorder}

If the semilattice L contains a **1** element, then the for-loop in the initialization of the procedure can be changed to

for j:=2 **to** n **do** A[j]:= **1 endfor**

Backward analysis problems can be similarly handled by changing the order in which the nodes of the flow graph are visited by the algorithm, that is, from n to 1 with the nodes numbered in rpostorder. The live variables bit-vector problem is in this category and can be modeled by the data flow framework LVBOT = (L, $\wedge$,F), where the elements of L are m-bit vectors (m is the number of different variables), the meet operation $\wedge$ is the logical bit-vector operation

OR. The zero element is the bit-vector of all 1's and the one element is the bit-vector of all 0's. The function space associated with L will consist of functions of the form $\langle p,v\rangle(x) = (x \text{ AND } p) \text{ OR } u$, where x is in L, p is the set of live variables preserved by the node associated with x, and u is the set of variables used in the node associated with x.

4.4 Other Examples of Data Flow Frameworks.

Two examples of bit-vector propagation frameworks, available expressions and live variables, were used in the previous section to ilustrate our discussion. We have also seen how constant propagation can be modeled by a semilattice framework. In this section we will mention some other examples of data flow frameworks discussed in the literature.

To solve common subexpressions elimination, Kildall [Kild73] uses an optimizing pool of expressions partitioned into equivalence classes. The meet operation is the intersection by classes and the operations on the pool consist of adding any expression to the partition which have operands equivalent to the one at the node being considered. This process is called **structuring**.

The structuring partition approach applied to solve loop invariant computations and indution variables have been discussed in [Fong75].

Type determination for very high level languages had been modeled by Tenenbaum [Tene74]. The elements of the lattice consist of sets of types which variables could assume. The meet operation is the union of sets of types and the functions associated with the nodes of the flow graph reflect certain inferences about types derived from the syntactic rules of SETL language.

A formal definition of the abstract interpretation of a program using a unified lattice model can be found in [Cous77a]. In later papers Cousot had used that approach for the discovery of invariant asertions of programs [Cous77b, Cous78].

CHAPTER V

INTERPROCEDURAL AND HIGH LEVEL DATA FLOW ANALYSIS

5.1 Interprocedural Data Flow Analysis.

The global data flow analysis methods discussed in the previous chapters assume that no procedure calls are present. Thus, these methods are called intraprocedural data flow analysis methods. In the presence of procedure calls data flow analysis becomes more complicated and new techniques, called interprocedural data flow analysis techniques, are required. Interprocedural data flow analysis has been neglected in most of data flow analysis papers but, in recent years, it has received increasing attention [Aho77b, Alle74, Bart77, Bart78, Harr77, Hech77, Lome77, Rose79, Shar77, Shar81, Whei80]. Several reasons may justify such attention. The first reason, in our opinion, is that intraprocedural data flow analysis is by now well understood and we are ready to proceed to more complicated problems. Second, interprocedural analysis has become more important because of the current emphasis on modularity on programs [Weih80]. Finally, the application of data flow analysis to other areas than optimization such as program verification, gives additional importance to its

study [Shar81]. Here we will try to look at some of the additional problems that procedure calls present to global data flow analysis and to briefly enumerate some approaches to their solution.

Intraprocedural data flow analysis assumes that the semantics of each program statement are immediately available, but this is not true in the case of a procedure call statement. The effects of the called procedure on the variables are not known without an examination of the procedure body. Traditionally, call statements have been treated either as black boxes, or the 'worst assumptions' have been made about the procedure effects; in the first case interprocedural analysis stops at any procedure call; with the worst assumptions relevant data flow information is usually lost.

The aim of the interprocedural data flow analysis is to make available the effects of each procedure call to global data flow analysis. Usually a 'summary' is associated with each procedure call; this summary may contain the following information [Bart77]:

(i) variables which can be modified.

(ii) variables used.

(iii) variables preserved.

The calling relationship among procedures may be represented by a directed graph named "call graph". Formally, a call graph is a directed graph G =(N,E), where

each node x in N represents a procedure of the program and an edge (x,y) in E represents one or more references in the procedure represented by node x of the procedure represented by node y. Using the call graph, an optimizing compiler should compute the summary information for each procedure and then proceed to global data flow analysis.

5.1.1 Some Interprocedural Analysis Problems.

Barth [Bart77] mentions three problems which make gathering of summary information difficult:

(a) In nonrecursive programs, call levels can be handled by analyzing the procedures in 'reverse invocation order' obtaining the summary of each called procedure before their calling procedure is analyzed. But there is no ordering with such a property in the case of recursive programs.

(b) The determination of 'aliases' is not a trivial problem. Aliases are variables which share the same location of memory and this problem may be found in programming languages with name or reference parameters.

(c) Correct treatement of variables in recursive programs also present difficulties since variables may be modified at different incarnations of a procedure.

Weihl [Weih80] discusses the following situation which adds difficulty to interprocedural flow analysis:

(d) The presence of procedure variables requires additional computation to find the call graph, and the presence of label variables requires the computation of their range for the construction of the control flow graph.

5.1.2 Some Approaches to Interprocedural Data Flow Analysis.

Several approaches to deal with procedure calls in global data flow analysis may be considered. An enumeration of such approaches is found in [Hech77].

(a) The worst case method which consits in making the most pessimistic assumptions about the effects of call statements on data flow information. Although some useful information about data flow may be lost using this method, it is applicable in the case of external procedures when their summary is not available.

(b) Procedure integration consists in incorporating the called procedure body into the calling procedure, renaming local variables if necessary , and replacing formal parameters by the actual arguments. Standard data flow analysis can then be performed.

(c) The one-pass method analyzes each procedure only once. This approach makes extremely conservative 'worst case' assumptions about the outside information reaching each procedure since each one is analyzed before their

calling procedure.

(d) The multi-pass method starts with an estimate of the unknown information about the effects of procedure calls and iterates it to make the required adjustements until the information stabilizes.

(e) Symbolic flow analysis gathers summary information by evaluating the expressions for each call in a symbolic manner.

(f) Mixed-strategy method uses a combination of the previous methods as procedure integration of short procedures and multi-pass method to solve the remaining calls.

Two more techniques are discussed in [Shar81]

(g) Functional approach which considers procedures as program blocks and the summary information is stablished through input-output relations for each block.

(h) The call string approach mixes together interprocedural and intraprocedural flow analysis. Interprocedural flow is made explicit by 'tagging' the information with the history of the procedure calls affecting it.

5.2 Very High Level Languages Optimization.

Recently, several papers about optimization of

programming languages of very high level have appeared in the literature. These papers are related to the SETL project at New York University [Tenn74, Schw74, Fong76, Fong77]. SETL is a set-oriented language and its vision of data as sets and mappings arises new problems from the data flow analysis point of view. To illustrate some of these problems we should mention the type determination problem [Tene74]. In languages without type declarations, type determination is usually an overhead run-time task. The data flow analysis may help to determine variables type at the compile time and [illegible] to obtain a more efficient code.

In [Fong77] an approach to common subexpression elimination in set-oriented languages is presented. The usual answer 'yes' or 'not' to the availability of an expression is not enough in very high level languages. For example, if after the computation of $C=A \bigcup B$ there is an "incidental assignment" $A=A \bigcup \{x\}$, we can say that C is partially available since $C=C \bigcup \{x\}$ can be easily recomputed. We say, then, that C was 'incidentaly kill' or 'wounded' [Rose81]. The idea is to keep information about the 'degree of availability', IAVAIL(n), of each expression R at each node n of the flow graph. The degree of availability may take values in the interval [0,1] where 1 is the degree of availability after generation, and 0 after killing. IAVAIL(n) is 1 for an expression R if R is partially available along each path m to n and there is a bound in the number of wounds. If IAVAIL(n) is 1, then

healing the wounds is cheaper than recomputing the expression from scratch. The IAVAIL information does not fit the standard semilattice-theoretic framework but it may be adapted to work in the case of reducible flow graphs [Fong77]. This problem is also discussed in [Rose81].

5.3 High Level Data Flow Analysis.

The methods discussed so far assume a low level representation of the program such as three address intermediate code. However, some work has been done recently about optimization at the source representation level. This is useful, for example, in source-to-source program transformations. The data flow analysis which precedes such transformations is called 'high level data flow analysis'. This analysis is done on the parse tree representation of the structures present in the source program. One technique used in high level data flow analysis is named 'method of attributes' [Babi78]. The attributes associated with each nonterminal are propagated along the parse tree by the computation of semantic equations associated with each programming language rule.

One advantage of 'high level data flow analysis ' is that a change in the program structure limits the data flow updating to the area where the change actually ocurred [Kenn81]. Rosen [Rose77] envisages in the method the creation of a 'modern ambitious compiler' which could be

used interactively for program definition. We will not expand high level data flow analysis any further here. A discussion of the topic may be found in the already cited references in this section.

5.4 Graph Grammars.

We will mention now a similar approach which also takes advantage of the program structure.The control flow graphs of structured programs appear to fall in a restricted subclass of general graphs. Each flow graph generally is made up of basic flow graphs representing the language statements. This fact makes possible the construction of graph grammars. The parsing of a flow graph using graph grammars rules allows to uncover loops and other control structures in a more natural way than using the interval method. Data flow analysis is then facilitated by this approach but its applicability is limited to certain classes of flow graphs [Kenn77, Kenn81].

CHAPTER VI

METHODS EVALUATION AND CONCLUSIONS

The main ideas about data flow analysis and the different methods to perform such analysis have been discussed in the previous chapters. However, nothing has been said so far about the efficiency and suitability of those methods to solve real problems. The most important measure of complexity of an algorithm is the 'time-complexity' which is the time needed to solve that problem expressed as a function of the size of the problem [Aho74]. To specify that time, we may count the number of steps required by the algorithm to process a given input. Time-complexity has been determined for most of the algorithms discussed in the literature, however, it is not easy to draw a quick conclusion of these studies. Each of the algorithms is limited to certain classes of problems or classes of flow graphs. The efficiency can also depend on the order in which the nodes of the flow graph are scanned. In this chapter we will present the time-complexity of certain algorithms along with some conclusions.

6.1 Node Ordering and Algorithm Efficiency.

Depending on the order in which the nodes of the flow graph are visited, different versions of data flow analysis algorithms have appeared. Since the efficiency of these versions can differ, we give now a brief explanation of the basic node orderings which are used by the algorithms.

Reverse postorder (rpostorder) and interval order were already mentioned in Chapters 2 and 3 respectively. These node orders are such that the algorithms using them visit each node after all its predecessors have been visited. Each node appears only once and they are considered 'reasonable' node orders [Hech77]. Iterative algorithms visit each node at every iteration until the data flow information stabilizes.

In [Kenn75] a node order called node listing is discussed. A node listing tells the data flow analysis algorithm to which nodes and in which order an equation is to be applied for data flow information propagation. A node listing may have the nodes repeated, but the algorithm stops after all the listed nodes are visited. Thus some possible unnecessary visits of the previous orders are eliminated using node listing and faster algorithms result. Node listing is useful for problems in which the information needs only to be propagated along cycle-free paths as in reaching definitions, live variables, and available expressions. But the computation of efficient node listing

is nonobvious in general [Hech77]. For reducible flow graphs, Aho and Ullman [Aho75] have shown that a node listing of length proportional to n log n (i.e. O(n log n)) may be constructed in time O(n log n) if the number of edges is of O(n), where n is the number of nodes of the flow graph.

6.2 Algorithms and their Time-complexity.

As mentioned above the time-complexity of an algorithm is specified by the number of steps required by the algorithm to process a given input. What is considered as a step depends on the operations performed by the algorithm. Data flow analysis algorithms generally propagate information by performing logical operations AND, OR, NOT on bit vectors. Each logical 'bit vector operation' may be considered as a step, named 'extended step'. Operations involving finite amounts of data and ordinary arithmetic operations are considered 'elementary steps'. In this sense, a logical operation on bit vectors of length m takes m elementary steps [Ullm73].

In [Ullm73] we can find several versions of iterative algorithms for the elimination of common subexpressions. First, a 'tabular' version is presented which requires O(me) elementary steps, where m is the vector length (number of expressions), and e is the number of edges in the flow graph. This version requires O(mn) steps in the case of a

'program flow graph', that is , a flow graph with n nodes and in which no node has more than two successors. Using bit vector operations, the algorithm takes O(ne) extended steps in a flow graph and at most $O(n^2)$ extended steps in a program flow graph.

Another algorithm using 3-2 balanced trees is dicussed in the same paper. This algorithm is applicable to reducible flow graphs and requires O(n log n) extended steps. A different algorithm, for the same class of graphs and with the same bound, is discussed in [Grah75].

In [Hech75], Hecht and Ullman describe a 'round-robin' iterative algorithm which uses 'rpostorder' node ordering and requires at most (d+2)(e+n) extended steps, where d is the largest number of back edges found in any cycle free path in the flow graph. Using 'extended' basic blocks, Kennedy [Kenn76] presents a similar algorithm which requires at most 2e(d+2) extended steps. These algorithms may require $O(n^2)$ extended steps in the worst case. In fact, d can be O(e), and e usually is O(n) [Hech77].

Kennedy [Kenn75] gives an iterative 'node listing' algorithm which takes O(n log n) steps to solve live variables problem.

Using the interval approach, Kennedy [Kenn76] proves that interval algorithms generally require less bit vector operations than iterative algorithms, but it is still $O(n^2)$ in the worst case.

It was already mentioned that the speed of the

algorithms may vary depending on the node ordering of the flow graph and that they may be restricted to a specific class of graphs or problems. Other factors that we should not forget are the programming complexity of the method and the data structures required. The interval method, for example, needs to keep and update several tables describing the derived sequence of the flow graph. A detailed ilustration of such tables can be found in [Hech77]. To put everything together, we reproduce here the summary given by Kennedy [Kenn81] adapted to our discussion. The letters S, M, and C in the third column stand for simple, medium and complex respectively. The column header 'both ways' refers to forward/backward propagation problems.

Table 6.1 - Complexity of algorithms.

Method	time com-plexity	design complex.	structure required	graph class	both ways
Iter.-'tabular'	n^2	S	no	all	yes
" -'round-r.'	n^2	S	no	all	yes
" -'node-lis.'	n log n	M	no	redu.	yes
" -'3-2 trees'	n log n	C	yes	redu.	no
Interval	n^2	M	yes	redu.	yes
Grammar	n	M	yes	L(gram.)	yes
High level	n	S	yes	parse-t.	yes

As we can see, there is not a general and best way to solve data flow analysis problems. A reduction in time-complexity usually requires additional data structures and an increase in programming complexity.

6.3 Optimization and Compiler Environment.

As mentioned in chapter I, section 1.2, the transformations which can be done to improve a computer program are numerous. The algorithms and methods to gather information necessary for such transformations are numerous also. But we may ask ourselves how many of these transformations are required of a compiler. There are several important factors to be considered in order to answer this question. Among the most important factors we should mention:

- desired complexity of the compiler;
- environment in which the compiler is to operate;
- the type of programming language;
- area of application of the programming language.

To produce better code requires a more sophisticated compiler, thus, the desired complexity must be determined. More sophistication generally means a more complex design and a slower compiler. Good optimization techniques require to analyze the produced code which therefore does not allow fast one-pass compilation.

In an academic environment the compiler may face heavy load, but the programs are going to run only once most of the time. Such a compiler should be able to provide very good error diagnostics but should not do much of code optimization. This philosophy may be found in the design of

the WATFOR compiler [Cowa70], the PL/C compiler [Conw73] and DITRAN compiler [Moul67]. These compilers emphasize more error diagnosis and error recovery both during the compile time and the run time. In fact the programmer generally needs to make several tries before his program is error free and in an academic environment the compilers handle most of the time undebugged student programs. On the other hand, programs which are used often should be optimized. At most non-academic installations some programs are used very heavily and they, therefore, should be improved by the compiler.

The optimizations required may depend on the source language. Operator strength reduction, for example, is important in algebraic languages as FORTRAN, but may not be necessary at all in set-oriented languages.

System software such as compilers and operating systems written in a high level language is another example of very often used programs whose object code should be optimized. Similarly, programs used in real time operations should be improved as much as possible and emphasis should be put on the optimization phase of the compiler.

Another factor which will help in the design of a more efficient compiler is to have a knowledge of the nature of the programs to be compiled. After studying a sample of FORTRAN programs, Knuth [Knut71] found that most of the programming is made of a small number of basic patterns and that a small percentage of a program (about 4 per cent),

generally accounts for more than 50 per cent of its execution time. The compiler thus could limit the expensive optimizations to a small percentage of the program. Still another interesting result of Knuth's study is that programs run four or five times faster when they are well optimized compared to their translation without optimization. This improvement was obtained making ,by hand, different types of transformations including constant propagation, invariant expression removal,strength reduction, test replacement and load-store motion as well as machine-dependent optimizations. Knuth also concludes that the existing optimizations techniques seem to be good enough since an attempt to optimize further is not worth while given the little difference in code improvement.

Wortman, Khaiat and Laskar [Wort76] compared six PL/I compilers, Optimizer, PL/I (F), PL/C, CHECKOUT, SP/K and PLUTO, and found that the production compilers (Optimizing and PL/I (F)) generate faster code than diagnostic compilers (PL/C, CHECKOUT, SP/K an PLUTO) and that, in general, statements compiled by the Optimizing Compiler ran 20 per cent faster. Another study of five ALGOL compilers [Wich72] shows that the speed of the code generated by a compiler depends also on the architecture of the computer.

The designer of a compiler then needs to take in account the environment in which the compiler is going to run. In practice, the most popular languages and their compilers are used in any environment. The designer can

always provide the global optimizations desired and allow to enable or to disable them by the setting of parameters at the time of compilation. This is the case with the CDC Fortran compiler [CDC81] and with the IBM PL/I (F) compiler [IBM70]. With the CDC FORTRAN compiler, for example, the user may choose the degree of optimization required by setting up the parameter OPT with a value ranging from 0 to 3. If the user chooses level 0 then the compiler evaluates constant subexpressions, eliminates redundant instructions and expressions within a statement and determines critical paths (PERT) to utilize the multiple functional units efficiently. Level 1 does the transformations of level 0 and local transformations. Global transformations are done when OPT has a value of 2; among them are live analysis,loop independent subexpressions moved out of the loop, operator strength reduction, array addresses and subscript expressions kept in registers during loop execution and indexed array references prefetched after safety checks in small loops. With level 3 the compiler will do unsave optimizations such as prefetching indexed array references without safety checks and making optimistic assumptions about the effects of external function execution on the the contents of registers. In the case of PL/I (F) compiler the user may control the optimization for a particular compilation in a similar way. Under level 2 the compiler does loop and subscript optimization. With other existent compilers the user is only allowed to choose optimization,

or no optimzation at all, by the use of a command switch at the compilation time. Such is the case with the DEC Cobol 68 compiler [DEC79] and the Decsystem10 FORTRAN compiler [DEC77]. When the optimization is enabled the FORTRAN compiler does several global optimizations; among those optimizations are elimination of redundant computations, reduction of operator strength, removal of constant computations from loops, constant propagation, removal of inaccessible code, global register allocation, I/O optimization, uninitialized variable detection and test replacement. The BLISS compiler as described by Wulf [Wulf75] does also several improvement transformations during the optimization phase (DELAY). Most of them are machine dependent transformations. Among others, we can mention: register allocation by determining the evaluation order of expressions, conversion of multiply to shift operations, and distribution of multiplications across additions. It also performs constant propagation.

To make the compiler task easier and obtain better results, the programmer should be familiar with what the compiler does to optimize the code and try to avoid certain constructs or bad programming habits. The best conceivable code may not be produced without certain interaction between the programmer and the compiler. Interactive editing, frequency counts, and tracing capabilities are also very desirable in a compiler.

Before finishing this section we should note that the

order of the optimizing transformations is important. Dead code elimination, for example, should follow constant propagation since the last transformation may produce dead instructions. The analysis of data flow should, then, procede in the required order too. The choice of the data flow analysis method and algorithms depends, once again, on different factors. Design complexity, type of optimization, language under consideration, and data flow analysis level are among those factors.

6.4 Other Applications of Data Flow Analysis.

So far the aim of the discussion of data flow analysis has been code optimization in compiling. But there are other applications which may need data flow information and which will find data flow analysis techniques useful. An area which can take advantage of these techniques is 'software engineering'. Although an analysis of such applications is not in the scope of this report, we mention some of them to indicate a larger applicability of data flow analysis. Among those applications are [Fosd76, Oste81] :

(a) Program testing. This technique consists in finding errors in a program. Data flow analysis allows to discover the static characteristics of a program. By contrast with dynamic testing, program errors can be found by data flow analysis without executing the program being

tested. Variables never used (dead) after been defined and uninitialized variables, for example, may be detected by live variables data-flow analysis.

(b) Program verification. Program verification is the process of demostrating the absence of errors, and data flow analysis techniques offer also certain program verification capabilities. For example, we can assert that a program does not have 'dead' variables if there are no variables which are both defined and not live at any node of the flow graph.

(c) Program documentation. Information about the structure and the functioning of the program can be generated by software tools. This requires certain information of the control and data flow of the program and such information may be obtained by data flow analysis.

6.5 Conclusions

Data flow analysis has received increasing attention during the last ten years and a quite extensive literature is already available about the subject. A brief overview of the topic has been presented in this report, where after an introductory chapter, the different methods of data flow analysis have been discussed. The iterative method appears to be the most popular because of its simplicity. The interval method may perform better than the iterative method, but is more complex to implement because it requires

extra data structures and because of the existence of irreducible flow graphs [Hech75].

The attempt to find a unified approach to data flow problems has led to the formulation of data flow analysis as a semilattice-theoretic framework [Kild73] and a well founded theory has appeared.

The main objective of data flow analysis is to gather the information necessary to improve computer compiled programs. Intraprocedural data flow analysis is already well understood and more attention is now being directed to interprocedural data flow analysis. Procedure calls make data flow analysis more difficult, however modularity and structured programming is necessary in the development of good software. Thus, interprocedural data flow analysis is now an urgent problem.

The special problems of the propagation of data flow information presented by very high level languages is another area of recent research. Recent papers consider data flow analysis at high level using the parse-tree representation of the program rather than a low level representation. Similarly, graph grammars is another high level approach which makes use of the basic constructs of structured programming.

The use of data flow analysis in other applications as software engineering, has increased the importance of the topic. In spite of all that research, the problems to be solved in data flow analysis are still numerous. Many of

the algorithms and solutions presented in the literature are restrictive and difficult to implement in practice, where the programs are more complicated than the simplified examples used in the literature [Rose81].

REFERENCES

Aho73 Aho, A.V., and Ullman, J.D. [1973]. The Theory of Parsing, Translation, and Compiling. V. 1, Prentice-Hall, Englewood Clifs, N.J.

Aho74 Aho, A.V., Hopcroft, J., and Ullman, J.D. [1974]. The Design and Analysis of Computer Algorithms. Addison- Wesley, Reading, Mass.

Aho75 Aho, A.V., and Ullman, J.D. [1975]. Node Listing for Reducible Flow Graphs. 7th. ACM Symposium on Theory of Computing, pp. 177-185.

Aho77a Aho, A.V., Johnson, S.C., and Ullman, J.D. [1977]. "Code Generation for Expressions with Common Subexpressions". J. ACM, 24:1, pp. 146-160.

Aho77b Aho, A.V., and Ullman, J.D. [1977]. Principles of compiler design. Addison-Wesley.

Alle70 Allen, F.E. [1970]. "Control flow analysis". SIGPLAN Notices, 5:7, 1-19.

Alle71 Allen, F.E., and Cocke, J. [1971]. "A Catalogue of Optimizing Techniques" in Design and Optimization of Compilers. R. Rustin, Ed., Prentice-Hall.

Alle74 Allen, F.E. [1974]. "Interprocedural Data Flow Analysis". Proc. IFIP Congress 74, North-Holland Publishing Co., Amsterdam, Holland, pp. 398-492.

Alle76 Allen, F.E. and Cocke, J. [1976]. A program data flow analysis procedure". Comm. ACM, 19:3,

137-147.

Alle81 Allen, F.E., Cocke, J., and Kennedy, K. [1981]. "Reduction of Operator Strength", in Program Flow Analysis: Theory and Applications. Muchnick, S.S., and Jones, N.D. (Eds.), Prentice-Hall.

Babi78 Babich, W.A., and Jazayeri, M. [1978]. "The Method of Attributes for Data Flow Analysis. Part I: Exhaustive Analysis". Acta Inf., 10, pp. 245-264

Babi78 Babich, W.A., and Jazayeri, M. [1978]. "The Method of Attributes for Data Flow Analysis. Part II: Demand Analysis". Acta Inf., 10, pp. 265-272.

Back81 Backus, J. [1981] History of FORTRAN I, II, and III, in History of Programming Languages, Wexelblat, R.L. (Ed.). Academic Press, N. Y.

Bart77 Barth, J.M. [1977]. "An Interprocedural Data Flow Analysis Algorithm". Conf. Rec. 4th ACM Symp. on POPL, Los Angeles, Cali., pp. 119-131.

Bart78 Barth, J.M. [1978]. "A practical interprocedural data flow analysis algorithm". Comm. ACM, 21:9, pp. 724-736.

Blum67 Blum, M. [1967]. A Machine-independent Theory of the Complexity of Recursive Functions. J. ACM 14:2, pp. 322-336.

Bron76 Bron, C., and de Vries, W. [1976]. A PASCAL compiler for PDP11 mini-computers. Software Practice and Experience, 6:1, pp. 109-116.

Cart77 Carter, J.L. [1977]. "A Case Study of a New Code Generation Technique for Compilers". Comm. ACM, 20:12, pp. 914-920.

CDC81 Control Data Corporation. FORTRAN 5 Reference Manual. Sunnyval, California, 1981.

Cock70a Cocke, J. [1970]. "Global common subexpressions eimination". SIGPLAN Notices 5:7, pp. 20-24.

Cock70b Cocke, J., and Schwartz, J.T. [1970]. Programming Languages and Their Compilers. Courant Institute of Mathematical Science, New York, N.Y.

Cock77 Cocke, J., and Kennedy, K. [1977]. "An Algorithm for Reduction of Operator Strength". Comm. ACM, 20:11, pp. 850-863.

Conw73 Conway, R.W., and Wilcox, T.R. [1973]. Design and implementation of a diagnostic compiler for PL/I. Comm. ACM 16:3, pp.169-179.

Cous77a Cousot, P., and Cousot, R. [1977]. Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. Conf. Rec. of 4th Symposium on POPL, pp. 238-252.

Cous77b Cousot, P., and Cousot, R. [1977]. Automatic Synthesis of Optimal Invariant Assertions. Mathematical Foundations. SIGPLAN Notices, 12:8, pp. 1-12.

Cous78 Cousot, P., and Halbwachs [1978]. Automatic Discovery of Linear Restraints Among Variables of a

Program. Conf. Rec. of 5th ACM Symposium on POPL, pp. 84-97.

Cowa70 Cowan, D.D., and Graham, J.W. [1970]. Design Characteristics of the WATFOR Compiler. SIGPLAN Notices, 5:7, pp. 25-36.

DEC79 Digital Equipment Corp. [1979]. COBOL 68 Language Reference Manual. Malboro, Massachusetts.

DEC77 Digital Equipment Corp. [1977]. FORTRAN Programmers Reference Manual. Maynard, Massachusetts.

Fong75 Fong, A.C., Kam, J., and Ullman, J.D. [1975]. "Application of Lattice Algebra to Loop Optimization". Conf. Rec. 2nd ACM Symp. on POPL, Palo Alto, Cali., pp. 1-9.

Fong76 Fong, A.C., and Ullman, J.D. [1976]. "Induction Variables in Very High Level Languages". Conf. Rec. 3rd ACM Symp. on POPL, Atlanta, GA, pp. 104-112.

Fong77 Fong, A.C. [1977]. "Generalized Common Subexpressions in Very High Level Languages". Conf. Rec. 4th ACM Symp. on POPL, Los Angeles, Cali., pp. 48-57.

Fosd76 Fosdick, L.D. and Osterweil, L.J. [1976]. "Data flow analysis in software reliability". Computing Surveys 8:30, 305-330.

Grah76 Graham, S.L. and Wegman, M. [1976]. "A fast and usually linear algorithm for global flow analysis".

J. ACM 23, pp. 172-202.

Grat71 Gratzer, G. [1971]. Lattice Theory: First Concepts and Distributive Lattices, W.H. Freeman and Co., San Francisco, Calif.

Harr77 Harrison, W.H. [1977]. "A New Strategy for Code Generation - The General Purpose Optimizing Compiler". Conf. Rec. 4th ACM Symp. on POPL, Los Angeles, Cali., pp. 29-37.

Hech72 Hecht, M.S. and Ullman, J.D. [1972]. "Flow graph reducibility". SIAM J. Computing, 1:2, pp. 188-202.

Hech74 Hecht, M.S. and Ullman, J.D. [1974]. "Characterizations of reducible flow graphs". J. ACM 21:3, 367-375.

Hech75 Hecht, M.S. and Ullman, J.D. [1975]. "A simple algorithm for global data flow analysis". SIAM J. Computing 4:4, 519-532.

Hech77 Hecht, M.S. [1977]. Flow Analysis of Computer Programs. Elsevier North-Holland, New York, N.Y.

Holl80 Holley, L.H., and Rosen, B.K. [1980]. "Qualified Data Flow Problems". Conf. Rec. 7th ACM Symp. on POPL, Las Vegas, Nevada, pp.68-82.

Hopc73 Hopcroft, J. and Tarjan, R.E. [1973]. "Efficient algorithms for graph manipulation". Comm. ACM 16, 372-378.

Horn74 Horning, J.J. [1974]. What the compiler should tell the user, in Bauer and Eickel, pp. 525-548.

IBM70 IBM Corp. [1970]. PL/I (F) Language Reference Manual. White Plains, New York.

Jone81 Jones, N.D., and Muchnick, S.S. [1981]. "Flow Analysis and Optimization of LISP-like Structures", in Program Flow Analysis: Theory and Applications. Muchnick, S.S., and Jones, N.D. (Eds), Prentice-Hall, Englewood Clifs, N.J.

Kam76 Kam, J.B. and Ullman, J.D. [1976]. "Global data flow analysis and iterative algorithms". J. ACM 23:1, 158-171.

Kam77 Kam, J.B. and Ullman, J.D. [1977]. "Monotone data flow analysis frameworks". Acta Informatica 7:3, 305-318.

Kapl78 Kaplan, M.A., and Ullman, J.D. [1978]. "A General Scheme for the Automatic Inference of Variable Types". Conf. Rec. 5th ACM Symp. on POPL, Tucson, AZ, pp. 60-75.

Kenn71 Kennedy, K. [1971]. "A global flow analysis algorithm". Int. J. Comp. Mahth. 3, 5-15.

Kenn75 Kennedy, K.W. [1975]. "Node listings applied to data flow analysis". Proc. 2nd ACM Symposium on Principals of Programming Languages, 10-21, ACM, New York.

Kenn76 Kennedy, K. [1976]. "A comparison of two algorithms for global data flow analysis. SIAM J. Computing 5:1, 158-180.

Kenn77 Kennedy, K., and Zucconi, L. [1977]. "Applications

of a Graph Grammar for Control Flow Analysis". Conf. Rec. 4th ACM Symp. on POPL, Los Angeles, Cali., pp. 72-85.

Kenn81 Kennedy, K. [1981]. "A Survey of Compiler Optimization", in Program Flow Analysis: Theory and Applications. Muchnick, S.S., and Jones, N.D. (Eds). Prentice-Hall, Englewood Cliffs, N.J.

Kild73 Kildall, G.A. [1973]. "A unified approach to global program optimization". Proc. ACM Symp. on Pr. of Programming Languages, pp. 194-206.

Knut71 Knuth, D.E. [1971]. "An Empirical Study of Fortran Programs". Software, Practice, Experience, 1:2, pp.105-133.

Knut73 Knuth, D.E. [1973]. The art of computer programming. V.1 (Fundamental algorithms, 2d Ed.), Addison Wesley Publ. Co., Reading, Mass.

Kou77 Kou, L.T. [1977]. "On live-dead analysis for global data flow problems". J. ACM, 24:3, pp. 473-483.

Lome77 Lomet, D.B. [1977]. "Data flow analysis in presence of Procedure calls". IBM J. RES. DEVELOP., 21:6, pp. 559-571.

More79 Morel, E. [1979]. "Global Optimization by Suppression of Partial Redundancies". Comm. ACM, 22:2, pp. 96-103.

Moul67 Moulton, P.G., and Muller, M.E. [1967].'DITRAN' a compiler enphasizing diagnostics. Comm. ACM 10:1,

pp.45-54.

Oste81 Osterweil, L. [1891]. Using Data Flow Tools in Software Engineering, in Program Flow Analysis: Theory and Applications. Muchnick, S.S., and Jones, N.D. (Eds), Prentice-Hall, Englewood Clifs, N.J.

Rose77 Rosen, B.K. [1977]. "High Level Data Flow Analysis". Comm. ACM, 20:10, pp. 712-724.

Rose78 Rosen, B.K. [1978]. "Monoids for rapid data flow analysis". Fifth ACM Symp. on Pr. of Programming Languages, pp. 47-59.

Rose79 Rosen, B.K. [1979]. "Data flow analysis for procedural languages". J. ACM, 26:2, pp. 322-344.

Rose81 Rosen, B.K. [1981]. Degrees of Availability as an Introduction to the General Theory of Data Flow Analysis, in Program Flow Analysis: Theory and Applications. Muchnick, S.S., and Jones, N.D. (Eds). Prentice-Hall, Englewood Clifs, N.J.

Russ76 Russel, D.L., and Sue, J.Y. [1976]. Implementation of a PASCAL compiler for the IBM 360. Software, Practice and Experience 6:3, pp. 371-376.

Scha73 Schaefer, M. [1973]. A mathematical theory of global program optimization. Prentice Hall Inc., Englewood Cliffs, N.J.

Schn73 Schneck, P.B., and Angel, E. [1973]. A FORTRAN to FORTRAN optimizing compiler. Computer J. 14:4, pp. 322-330.

Shar81 Sharir, M. [1981]. "Two Approaches to

Interprocedural Data Flow Analysis" in Program Flow Analysis: Theory and Applications. Muchnick, S.S., and Jones, N.D. (Eds), Prentice-Hall, Englewood Clifs, N.J.

Tarj72 Tarjan, R.E. [1972]. "Depth-first search and linear graph algorithms". SIAM J. Computing 1:2, 146-160.

Tarj74 Tarjan, R.E. [1974]. "Testing Flow Graph Reducibility". Journal of Computer and System Science, 9:3, pp. 355-365.

Tarj76 Tarjan, R.E. [1976]."Iterative Algorithms for Global Flow Analysis" in Algorithms and Complexity, New Directions and Recent Results, ed. Traub, J.F., Academic Press, N.Y., pp. 11-101.

Tarj81a Tarjan, R.E. [1981]. "A Unified Approach to Path Problems". J. ACM, 28:3, pp. 577-593.

Tarj81b Tarjan, R.E. [1981]. "Fast Algorithms for Solving Path Problems". J. ACM, 28:3, pp. 594-614.

Tene74 Tenenbaum, A. [1974]. Type determination in a language of very high level. TSO - 3, Courant Inst., N.Y.U., N.Y.

Ullm73 Ullman, J.D. [1973]. "Fast algorithms for the elimination of common subexpressions". Acta Informatica 2, 191-213.

Ullm75 Ullman,J.D. [1975]. "A survey of data flow analysis techniques". Proc. 2nd USA-Japan Computer Conference, pp.335-342, AFIPS Press, Montvale, NJ.

Weih80 Weihl, W.E. [1980]. "Interprocedural Data Flow Analysis in the presence of pointers, procedure variables and label variables". Conf. Rec. 7th ACM Symp. on POPL, Las Vegas, Nevada, pp. 83-94.

Wich72 Wichman, B.A. [1972]. Five ALGOL compilers. Computer J. 15:1, pp. 8-12.

Wilh79 Wilhelm, R. [1979]. "Computation and use of data flow information in optimazing compilers". Acta Informatica, 12:3, pp.209-225.

Wort76 Wortman, D.B., Khaiat, P.J., and Laskar, D.M. [1976]. Six PL/I compilers. Software, Practice and Experience 1:4, pp. 309-333.

Wulf75 Wulf, W., et al. [1975]. The Design of an Optimizing Compiler. Elsevier North-Holland, New York.