A SURVEY OF DATA FLOW ANALYSIS TECHNIQUES

Ken Kennedy

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Typed by Martha J. Cooper
Formated using the Yorktown Formatting Language
Printed on the experimental printer

Abstract

Compiler optimization can be a tremendous benefit to high-level language programming because it compensates for some of the inefficiencies in compiler-generated code. But to be effective, most optimization techniques require global information about the definitions and uses of data within the program; this survey describes several important methods for gathering such information. Section 2 covers *value numbering,* a major technique for analyzing straight-line code. Section 3 describes and compares nine fast algorithms suitable for solving simple data flow analysis problems. Section 4 introduces *use-definition chains,* a method for efficiently handling more complex problems. *Symbolic interpretation,* a more general but less efficient method for complex problems, is treated in Section 5. Finally, the application of these techniques to very-high-level language optimization is discussed in Section 6.

A SURVEY OF DATA FLOW ANALYSIS TECHNIQUES

Ken Kennedy

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Compiler optimization can be a tremendous benefit to high-level language programming because it compensates for some of the inefficiencies in compiler generated code. But to be effective, most optimization techniques require global information about the definitions and uses of data within the program; this survey describes several important methods for gathering such information. Section 2 covers *value numbering,* a major technique for analyzing straight-line code. Section 3 describes and compares nine fast algorithms suitable for solving simple data flow analysis problems. Section 4 introduces *use-definition chains,* a method for efficiently handling more complex problems. *Symbolic interpretation,* a more general but less efficient method for complex problems, is treated in Section 5. Finally, the application of these techniques to very-high-level language optimization is discussed in Section 6.

1. INTRODUCTION

High-level programming languages are valuable programming tools because they permit the specification of algorithms in notations more natural for expressing the abstract concepts involved. Thus, freed from attending to numerous machine-dependent implementation details, the programmer can produce correct, reliable code more easily. Why then aren't such languages universally used for programming? The usual answer is that the resulting programs are inefficient. That is, the code generated by a high-level language is less efficient than the code a good assembly language programmer would write. The problem is that the generality of programming languages, the very generality which is such a desirable aid to algorithm specification, prevents the programmer from making use of specific machine features to improve the efficiency of his code. Unfortunately, compilers for these languages fail to take up enough of the slack. Since a major aim of programming languages is to encourage programming at a more abstract level, there must be an improvement in the efficiency of object programs produced by compilers. This is the goal of compiler optimization.

Note that optimization is not intended to compensate for poor programming, but rather to reduce the inefficiencies in code to within "reasonable" bounds -- to a point where the advantages of high-level language programming outweigh any remaining efficiency penalties. For some languages, optimizing compilers might well be expected to produce code for inner loops that would be competitive with loops hand coded by assembly language programmers.

This last goal is difficult to achieve because high-level languages, if they are to be usable, must include general-purpose features flexible enough to serve many different applications. It is not enough to merely include a grab-bag of specialized features because programmers would find such a grab-bag difficult to learn and use. The assembly language expert can write efficient code because he knows the specific purpose to which each data structure in his program will be put; therefore he can choose for each structure the machine realization that will be most efficient. By contrast, the high-level language programmer must use one of the general-purpose data structures provided by the language. In the absence of better information, the compiler generates code for accesses to these structures which will be correct for any legal application. Thus it is unable to take advantage of any efficient short cuts which the specific problem at hand might allow. If the compiler is to compete with assembly language coding, it must be able to determine enough of the nature of the program being compiled to safely take those shortcuts; in other words, it must be able to perform some kind of global program analysis.

As an example, consider run-time subscript range checking. It is desirable to capture all attempts to reference outside the limits of an array because out-of-bounds references are the sources of many subtle errors. Unfortunately, range checks are expensive and can result in a significant speed degradation. Optimization offers a viable alternative to the common but questionable practice of eliminating all range checks: global program analysis can show that many range checks are superfluous, while others may be safely moved to less frequently executed code [Har75, Sul77]. The result will be more efficient programs without the cost of compromised reliability.

There is a widely-held notion that optimization is intended to compensate for bad programming. Nothing could be further from the truth. In fact, no currently-known technique can compensate for the main component of bad programming: a poor choice of algorithm. Instead, optimization encourages *good* programming by making high-level languages more attractive and by taking care of small matters of efficiency so the programmer is free to concentrate on the essence of his problem.

A variety of code improvement transformations have been proposed in the literature; I won't attempt to discuss them all since they are covered in two important compendia: The Allen-Cocke catalogue [AlC72a] and the "Irvine Catalogue" [Sta76]. But as background for the discussion of analysis methods, I will mention the most prominent techniques. First, two transformations are fundamental to optimization in straight-line code.

a. *redundant subexpression elimination* [Coc70, Fon77]:
If two instructions that both compute the expression $A*B$ are separated by code which contains no store into either A or B, then the second instruction can be eliminated if the result of the first is saved.

b. *constant folding* [CoS70]:
If all the inputs to an instruction are constants whose values are known, the result of the instruction can be computed at compile-time and the instruction replaced by a "load" of the constant value.

In simple loops, two more transformations can lead to significant improvements.

c. *code motion* [Coc70, CoS70]:
An instruction that depends only upon variables whose values do not change in a loop may be moved out of the loop, improving performance by reducing the instruction's frequency of execution.

d. *strength reduction* [All69, CoK77, FoU76, PaS77]:
Instructions that depend on the loop induction variable cannot be moved out of the loop, but sometimes they can be replaced be less expensive instructions. For example, in the loop

```
I:=1;
while I < 100 do
    •
    •
    A:= I*5;
    •
    •
    I:= I+1
od
```

the value of $I*5$ can be saved in a temporary T whose value is incremented by 5 on each iteration; $I*5$ can then be replaced by a load from T as shown below.

```
I:=1;
T:=5;
while I < 100 do
    •
    •
    A:= T;
    •
    •
    I:= I+1;
    T:= T+5
od
```

In effect, the multiplication has been replaced by an addition.

Automatic introduction of instructions at new positions in a program (à la code motion) gives rise to two important questions. First, the *safety* question asks whether the new instruction can cause an error interrupt that would not have occurred in the original program. This problem can be illustrated by the example in Figure 1. It is easy to see that if a computation of A/B is inserted at point p in block 1, the computation in block 3 becomes redundant and can be eliminated. But what if the purpose of the branch from block 2 to block 3 is to prevent an attempt to divide by zero? Moving A/B to block 1 might well introduce an error interrupt that the programmer has been careful to avoid.

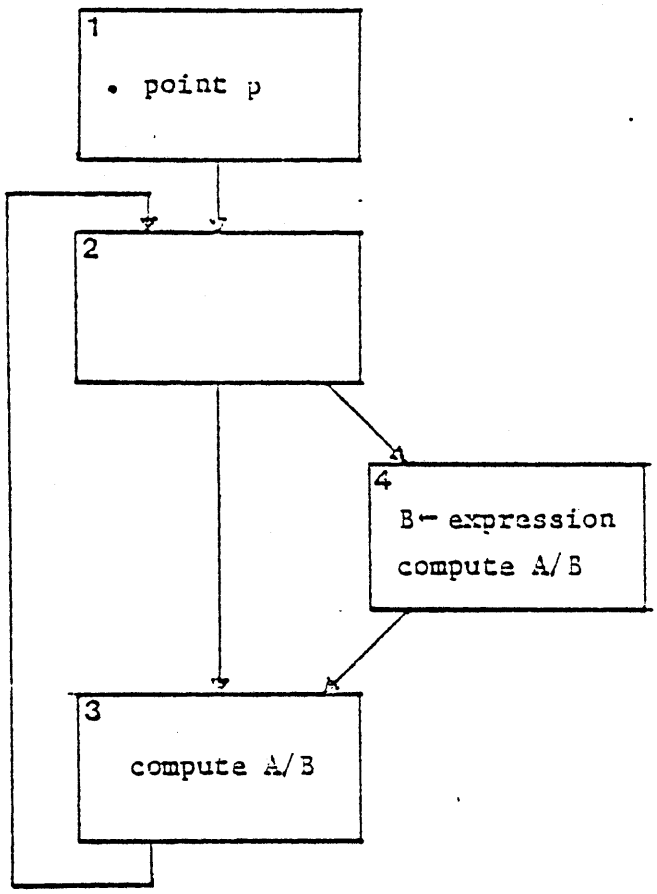


Figure 1. Safety example.

The question of *profitability* asks whether we are really moving code to a region of less frequent execution. Most compilers assume that code inside a loop is executed more often than code outside the loop, but this assumption could be wrong if there are several alternative branches within the loop. It is possible to do a fairly complete job of frequency estimation [CoK76], but few compilers make the attempt since it is not known whether the benefits will justify the cost.

Both "constant folding" and "redundant subexpression elimination", introduced earlier as local optimizations, can be applied on a global scale as well. Complementing these are two new global optimizations that "clean up" after other transformations.

e. *variable folding* [LoM69]:
Instructions of the form A:=B will become useless if B can be substituted for subsequent uses of A.

f. *dead code elimination* [Ken75]:
If transformations like variable folding are successful, there will be many instructions whose result is never used. Dead code elimination detects and deletes such instructions.

An extremely important class if transformations is intended to improve the efficiency of procedure invocation.

g. *procedure integration* [AlC72]:
Under certain circumstances, a procedure call can be replaced by the body of the procedure being called (open linkage); in other cases the overhead associated with standard calling sequences, parameters, and global variables can be reduced by compiling the procedure with the calling program (semi-open linkage).

Procedure integration is an extremely important optimization because procedure calls, desirable from the point of view of programming methodology, are often unbelievably inefficient. Thus good modular programming is penalized rather than rewarded by most compilers.

The last three optimizations are classified as "machine dependent" because they aim to increase efficiency by taking advantage of special features of the target machine.

h. *register allocation* [Bea74]:
This optimization seeks to eliminate load and store instructions by assigning variables to CPU registers whenever possible.

i. *instruction scheduling* [SeU70, Bea72]:
The proper arrangement of instructions often leads to improved performance. Different machines give rise to different scheduling criteria -- on a machine with pipelined arithmetic units, the goal is to achieve maximum parallelism, while on simpler machines the goal is to minimize register usage.

j. *detection of parallelism* [Sck75]:
For vector machines it is desirable to detect inherently parallel operations and code them as vector instructions.

This list is by no means complete, but it gives the flavor of some typical optimizing transformations. For those interested in reading further, and excellent introductory treatment of optimization appears in [AhU77], and Knuth's famous empirical study [Knu71] demonstrates the utility of various optimization techniques.

2. OPTIMIZATION IN BASIC BLOCKS

One of the first steps in analyzing a program for the purpose of code improvement is to subdivide the program into *basic blocks*, which are simply sequences of consecutive instructions that are always executed from start to finish. In other words, a basic block may only be entered at the first instruction and left at the last. Figure 2 shows how a PL/I program would be partitioned into basic blocks.

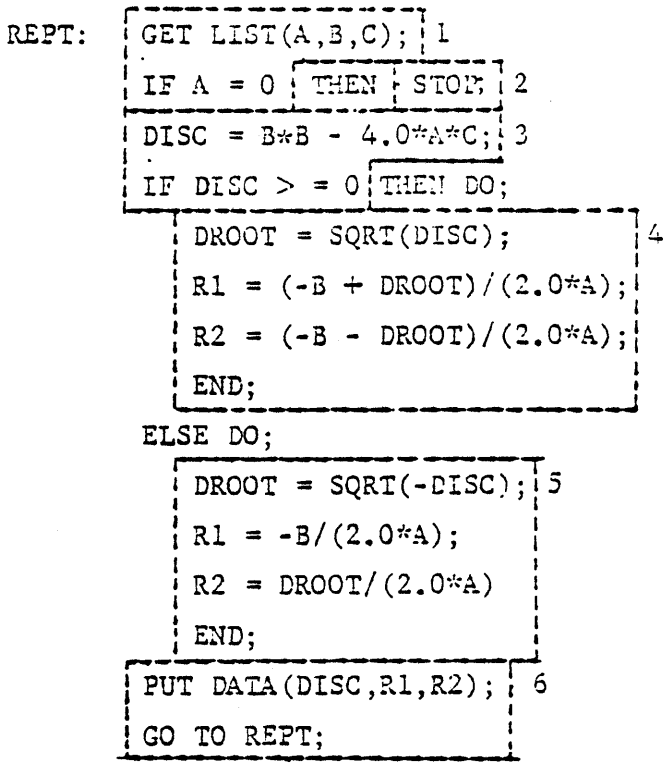


Figure 2. A PL/I program fragment partitioned into basic blocks.

Of course, in a compiler the partitioning is usually performed on some intermediate code representation of the program.

The subdivision process itself is fairly straightforward. I present a method adapted from [AhU77], that identifies a set of *leader instructions*, instructions which begin basic blocks, and then constructs a block by appending to its leader all subsequent instructions up to, but not including, the next leader. The algorithm is informally specified in an Algol-like high-level language which admits set theoretic notation.

Algorithm BB: Basic Block Partition
Input: A program PROG in which instructions are numbered in sequence from 1 to |PROG|. INST(i) denotes the ith instruction.
Output:
1. The set LEADERS of initial block instructions.
2. $\forall x \in$ LEADERS, the set BLOCK(x) of all instructions in the block beginning at x.

Method:

```
begin
   LEADERS := {1}; ¢ first instruction in PROG ¢
   for i := 1 to |PROG| do
      if INST(i) is a branch
         then add the index of each potential target to LEADERS
         fi
      od;
   TODO := LEADERS;
   while TODO ≠ ϕ do
```

```
        x := element of TODO with smallest index;
        TODO := TODO − {x};
        BLOCK(x) := {x};
        for i := x+1 to |PROG| while i ∉ LEADERS do
            BLOCK(x) := BLOCK(x) ∪ {i}
            od
        od
    end
```

Once the program is subdivided into blocks, each block can be optimized using local techniques. In this section I will describe the *value numbering* scheme of Cocke and Schwartz [CoS70], which performs redundant expression elimination and constant folding in straight-line code. As a side effect the method can also compute some of the information used by the global analysis methods treated later.

Suppose the source language version of a basic block under consideration is as follows:

```
A:= 4
K:= I*J+5
L:= 5*A*K
M:= I
B:= M*J+I*A
```

This might be transformed into the intermediate code in Table 1.

T1: A:= C4	*T5: C5*A*	*T9: M*J*
*T2: I*J*	*T6: T5*K*	*T10: I*A*
T3: T2+C5	*T7: L:= T6*	*T11: T9+T10*
T4: K:= T3	*T8: M:=I*	*T12: B:=T11*

Table 1. Intermediate code example.

Each triple in this code represents a simple operation; operands may be variables, constants (e.g., *C4*) or the results of previous operations (e.g., *T2*).

The main data structure of the "value numbering" method is a hash-coded *table of available expressions* which is used to help uncover redundant subexpressions. As each triple is treated in sequence from the start of a block, the table is searched for a previous instance of the same expression. If a match is found, the new triple may be eliminated if all subsequent references to it are replaced be references to the previous triple.

For the method to work, there must be some way to determine when two operands are identical. This is provided by a system of *value numbers* in which each distinct value created or used within the block receives a unique identifying number. Two entities have the same value number only if, based upon information from the block alone, their values are provably identical. For example, after scanning the first instruction in Table 1,

T1: A:= C4,

variable *A* and constant *C4* would have the same value number. The "current" value number associated with a variable (or constant) is kept in the symbol table entry for that variable; the value number for the result of a triple is kept in the table of available computations and as an auxiliary field of the triple itself. The hash function for entry to the available expression table is based on the value numbers of the operands and a special code for the operator.

Constant folding is handled via an auxiliary bit in each symbol table entry, indicating whether the current value is a constant, and a bit in each triple, indicating whether the result is a constant. Also required is a table of constants, indexed by value number, which contains the actual run-time values of constants.

Algorithm VN, presented in a high-level mixture of English and Algol, embodies the ideas discussed so far. Note that an instruction is assumed to be the value of a structured variable with an operator field OP, some auxiliary information and two operands L and R (left and right, respectively).

Algorithm VN: Value Numbering in a Basic Block
Input:
1. A basic block of triples.
2. A symbol table SYMTAB.

Output: An improved basic block, after redundant subexpression elimination and constant folding.
Intermediate:
1. Table of available expressions AVAILTAB.
2. Table of constants CONSTVAL.

Method:

```
begin
   while there is another instruction do
      INSTR := the next instruction;
      OPERATOR := OP of INSTR;
      if OPERATOR = store then
         find r, the value number of R of INSTR
            (this may assign a new value number);
         if r represents a constant value then
            so indicate in the SYMTAB entry for L of INSTR
            fi
      else ¢ an expression ¢
         find value numbers l,r for L of INSTR and R of INSTR
            (this may assign new value numbers);
         if l and r represent constant values then
            compute the value x of the result by applying OPERATOR to
               CONSTVAL(l) and CONSTVAL(r);
            enter the new constant x in CONSTVAL, assigning a new value number
               in the process;
            delete INSTR
         else ¢ check for availability ¢
            look up the triple <l,operator,r> in AVAILTAB, setting FOUND := true
               if successful;
            if FOUND then
               record the fact that any reference to this triple is to be subsumed by a
                  reference to the previous one (a pointer to which is contained in
                  AVAIL);
               delete INSTR;
            else ¢ not available ¢
               enter <l,operator,r> in AVAILTAB, assigning a new value number to
                  the result
      fi fi fi
   od
end
```

Consider the application of this algorithm to the example intermediate code from Table 1. In processing triples 1 through 4, nothing unusual takes place. Value numbers are assigned to variables *A*, *I*, *J* and *K* and to constants *C4* and *C5*. The results of triples *T2* and *T3* are recorded as available. The information collected up to this point is displayed in Table 2.

At instruction 5, the algorithm looks up *C5* and *A* and discovers that they are both constant. The resulting *C20* may be computed from values in CONSTVAL; it receives a new value number (7) and is recorded in CONSTVAL. Finally, triple 5 is deleted. In the next step, triple 6 will be modified to use *C20* in place of *T5*.

	Name	Value #	Constant?
1	C4	1	yes
2	A	1	yes
3	I	2	no
4	J	3	no
5	C5	5	yes
6	K	6	no

SYMTAB

	Result Value #	Constant?
T1	1	yes
T2	4	no
T3	6	no
T4	6	no

Auxiliary Fields of Triples

Value #	Value
1	4
5	5

CONSTVAL

Left Value #	OP	Right Value #	Result Value #	Original Instr.
2	*	3	4	T2
4	÷	5	6	T3

AVAILTAB

Table 2. Information collected up to instruction 5.

	Name	Value #	Constant?
1	C4	1	yes
2	A	1	yes
3	I	2	no
4	J	3	no
5	C5	5	yes
6	K	6	no
7	C20	7	yes
8	L	8	no
9	M	2	no

SYMTAB

	Result Value #	Constant?	
T1	1	yes	
T2	4	no	
T3	6	no	
T4	6	no	
T5*	7	yes	(deleted)

Auxiliary Fields of Triples

Value #	Value
1	4
5	5
7	20

CONSTVAL

Left Value #	OP	Right Value #	Result Value #	Original Instr.
2	*	3	4	T2
4	÷	5	6	T3
7	*	6	8	T6

AVAILTAB

Table 3. Information collected up to instruction 9.

Table 3 displays the information collected by the algorithm up to instruction 9. At this point it discovers that operands *M* and *J* have value numbers 2 and 3 respectively and that there is a previous computation (*T2*) of the product of these values. Therefore triple 9 can be deleted and subsequent references to it replaced by references to *T2*. The final optimized code is shown below.

T1: A:= C4
*T2: I*J*
T3: T2+C5
T4: K:= T3
*T6: C20*K*
T7: L:= T6
T8: M:= I
*T10: I*A*
T11: T2+T10
T12: B:= T11

It is especially interesting that instruction 9 is discovered to be identical to *I*J* even though an alias is used for *I*.

The method I have described is an elementary prototype of more sophisticated versions which can also handle array references and structured variables [CoS70, AhU77, KeZ78].

An important side effect if this or any other basic block analysis routine is that it can be modified to compute certain sets which are useful in determining global information. For example, the final version of the available computations table can be used to determine the set of expressions which are "available on exit" from the block. In the next section we turn to the problem of performing global analysis once we have such sets for each basic block.

3. GLOBAL DATA FLOW ANALYSIS

While analysis within basic blocks can lead to substantial improvements in a program, larger gains may be achieved by going a step further and gathering information on a global scale. For example, suppose the expression $A*B$ in block b is not eliminated by local methods; that is, there is no earlier computation of $A*B$ in b. Suppose also that neither A nor B is redefined in b prior to the computation of $A*B$. If we can prove that, no matter what control path is to be taken at run time, $A*B$ will always be computed before control reaches b, then we can still eliminate the computation in b. Establishing facts like this requires an analysis of control flow in the program that is thorough enough to yield useful information about data relationships.

In essence, the problem is this: given control flow structure, we must discern the nature of the data flow (which definitions of program quantities can affect which uses) within the program. The questions about data flow fall into two classes:

(1) Those which, given a point in the program, ask what can happen before control reaches that point (i.e., what definitions can affect computations at that point);

(2) Those which, given a point in the program, ask what can happen after control leaves that point (i.e., what uses can be affected by computations at that point).

Class 1 problems are usually called *forward flow* problems, while class 2 problems are *backward flow* problems. The gathering of information to solve problems of either class is accomplished in two phases. Once the program is subdivided into basic blocks, possible block-to-block transfers are noted and program loops are found. This phase is known as *control flow analysis*. Next the information about how uses and definitions relate to one another is gleaned in the *global data flow analysis* phase. The construction of data flow information is difficult because most nontrivial programs have complex control flow graphs; nevertheless, a number of solution methods exist. In this paper I shall outline a few of the most important.

The control flow of a program may be represented as a directed graph $G=(N,E,n_0)$ where N is the set of nodes, E is the set of edges and n_0 is the program entry node. In this model, nodes represent basic blocks and edges represent possible block-to-block transfers. Figure 3 shows the control flow graph corresponding to the PL/I program in Figure 2.

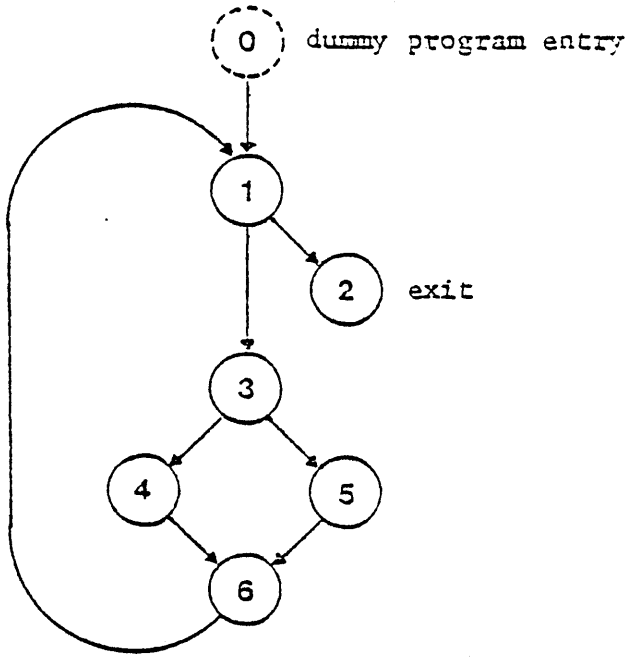


Figure 3. Control flow graph for Figure 2.

Two special notations will be used frequently in discussing control flow graphs. The *successor set S(x)* for a node x is defined as

$$S(x) = \{y \in N \mid (x,y) \in E\}$$

and the *predecessor set P(x)* is

$$P(x) = \{y \in N \mid (y,x) \in E\}$$

A *simple path* in G is a sequence of nodes $(n_1,n_2,\ldots,n_k)$ such that all nodes are distinct and $(n_i,n_{i+1}) \in E$, $1 \leq i \leq k$. A *simple cycle* is a simple path except that $n_1 = n_k$.

We shall use as examples two problems which are typical of class 1 and class 2 data flow problems.

a) Available Expression Analysis.

We say that an expression is *defined* at a point if the value of that expression is computed there. An expression is said to be *killed* by a redefinition of one of its argument variables. In these terms an expression is *available* at point p in G if every path leading to p contains a prior definition of that expression which is not subsequently killed. Let AVAIL(b) be the set of expressions available on entry to block b. We define a system of equations for AVAIL(b), $b \in N$, in terms of sets which can be computed from local information. Let NKILL(b) be the set of expressions which are not killed in block b and DEF(b) be the set of expressions which are defined in b without being subsequently killed in b, i.e., the set of expressions which are always available on exit from b. These definitions lead directly to the system of equations:

$$\mathrm{AVAIL}(b) = \bigcap_{x \in P(b)} (\mathrm{DEF}(x) \cup (\mathrm{AVAIL}(x) \cap \mathrm{NKILL}(x))) \qquad (*)$$

Solution of this system will provide the desired global information.

b) Live Variable Analysis

A path in $G=(N,E,n_0)$ is said to be *X-clear* if that path contains no assignment to the variable X. The variable X is *live* at point p in G if there exists an X-clear path from p to a use of X. Let LIVE(b) be the set of variables which are live on entry to block b. Once again we seek a system of equations for the live sets in terms of local sets. Let IN(b) be the set of variables which are live on entry to b because of a use within b, and let THRU(b) be the set of variables which are redefined in b. The following system of equations is the result.

$$\mathrm{LIVE}(b) = \mathrm{IN}(b) \cup \bigcup_{x \in S(b)} (\mathrm{THRU}(b) \cap \mathrm{LIVE}(x)) \qquad (**)$$

Similar equation systems can be developed for most data flow analysis problems. In fact, Kildall [Ki173], Kam and Ullman [KaU76], Graham and Wegman [GrW76], and Tarjan [Tar75b] all formalized their treatment of data flow analysis by providing axioms for "acceptable" equation systems, thus unifying their methods. To show that a particular problem can be handled by a standard algorithm, one need only show that the sets of quantities and rules for combining the sets at control flow junctions satisfy the required axioms. This approach simplifies the discussion of data flow methods. Curiously, it has also contributed to the classification of the algorithms by ranges of applicability [KaU76, Fon77]. Fast solution methods to these problems have taken a number of forms. Nine such methods are surveyed here, four in detail.

3.1 Iterative Techniques

Perhaps the simplest approach to data flow analysis is to iterate through the nodes to the graph applying the appropriate equations until no changes take place. Such a method has been studied by Hecht and Ullman [HeU75, Ull73] and subsequently by Kennedy [Ken76]. Here is the iterative algorithm for live variable analysis.

Algorithm IT: Iterative Live Analysis
Input: IN(b), THRU(b), $\forall b \in$ N.
Output: LIVE(b), $\forall b \in$ N.
Method:

```
begin
   for all b∈N do LIVE(b) := IN(b)od;
   change := true;
   while change do
      change := false;
      for all b∈N do
         oldlive := LIVE(b);
         LIVE(b) := IN(b) ∪ ⋃ (THRU(b)∩LIVE(x));
                           x∈S(b)
         if LIVE(b) ≠ oldlive then change := true fi
         od
      od
   end
```

If n = | N |, this algorithm requires $O(n^2)$ extended (or "bit vector") steps for the entire computation. Kildall [Ki173] has described a very general form of the iterative algorithm using

lattice theory while Kam and Ullman [KaU76] have shown that there exist optimization problems for which the iterative algorithm does not converge rapidly -- for example, constant propagation.

3.2 Nested Strongly-Connected Regions

A somewhat structured approach to data flow is based upon the loop organization in the program. This method proceeds from local to global analysis by first extending data flow information to inner loops, then effectively collapsing these loops to single nodes before continuing to the next level. Many optimizations such as code motion can be performed in stages using this method with code being "bubbled" outward to less frequently executed regions. This is the technique originally used by Allen [All69]. The difficulty is that it is not always easy to find a suitable collection of nested strongly-connected regions. The accepted way of locating such a collection was first devised by Earnest, Balke and Anderson [EBA72]; it involves the application of two ordering algorithms on the nodes of the control flow graph. Earnest [Ear74] continued this work by presenting a number of optimization algorithms which used nested regions. Beatty [Bea74] has developed an elegant register assignment algorithm using this method.

3.3 Interval Analysis

A simpler way to partition the control flow graph into regions was developed by Cocke and Allen [All70, All71, Coc70, AlC76]. An *interval* in G is defined to be a set I of blocks with the following properties:

(1) There is a node $h \in I$, called the *head* of I, which is contained in every control flow from a block outside I to a block within I; i.e., I is a single-entry region.
(2) I is connected. (This property is trivial if C is connected.)
(3) $I - \{h\}$ is cycle-free; i.e., all cycles within I must contain h.

Given a node h in some graph G, the following algorithm, due to Allen and Cocke [AlC76], constructs MAXI(h), the maximal interval with head h. In presenting the algorithm, I use the notation $S[M]$ where M is a set of nodes, to mean

$$\bigcup_{x \in M} S(x),$$

that is, the set of successors of nodes in M.

Algorithm MI: Maximum Interval Construction.
Input: The specified head h.
Output: MAXI(h).
Method:

```
begin
   I := {h};
   while ∃x ∈ (S[i] − I) such that P(x) ⊂ I
      do
         I := I ∪ {x}
      od;
   MAXI(h) := I
end
```

As we shall see, the order in which Algorithm MI adds nodes to an interval I is important, so it is usually given a name: *interval order*. Interval order is a total ordering on I which

preserves the partial order generated by the subgraph $I-\{h\}$. The significance is that if nodes of I are processed in interval order, a particular node $b(\neq h)$ will be treated only after every node in $P(x)$ has been processed. Similarly, if I is processed in *reverse interval order*, every node in $S(x)\cap I$ will be treated before x is. These order-of-processing observations are crucial to data flow algorithms based on intervals.

Using Algorithm MI as a subprogram, the following algorithm, also due to Allen and Cocke [AlC76], partitions a flow graph into a set of disjoint intervals. Algorithm IP is based upon the observation that any node which is the successor of some node in interval I, but which is not in I itself, must be the head of some other interval J.

Algorithm IP: Interval Partition.
Input: A flow graph $G=(N,E,n_0)$.
Output: A set INTS(G) of disjoint intervals which form a partition of G.
Auxiliary:
A set H of potential interval heads.
A set DONE of heads for which intervals have been computed.
Method:

```
begin ¢ the program entry n0 is a head ¢
   H := {n0};
   DONE := φ;
   while H ≠ φ do
      x := an arbitrary node in H;
      find MAXI(x) using Algorithm MI;
      INTS(G) := INTS(G) ∪ {MAXI(x)};
      ¢ add new heads ¢
      H := H ∪ (S[MAXI(x)] − MAXI(x) − DONE)
      od
   end
```

As an example, consider the flow graph displayed in Figure 4. When Algorithm IP is applied to this graph, it identifies nodes 1,2 and 5 as interval heads; the corresponding intervals are {1}, {2,3,4} and {5,6,7}

For a given flow graph G, the *derived flow graph* $I(G)$ is defined as follows:
(a) The nodes of $I(G)$ are the intervals in INTS(G).
(b) If J,K are two intervals, there is an edge from J to K in $I(G)$ if and only if there exist nodes $n_J\in J$ and $n_K\in K$ such that n_K is a successor of n_J in G. Note that n_K must be the header of K.
(c) The initial node of $I(G)$ is MAXI(n_0).

The sequence $(G_0,G_1,\ldots,G_m)$ is called the *derived sequence* for G if $G=G_0$, $G_{i+1}=I(G_i)$, $G_{m-1}\neq G_m$, and $I(G_m) = G_m$. G_i is called the *derived graph of order i* and $G_m$is the *limit flow graph* of G. A flow graph is said to be *reducible* if and only if its limit flow graph is the trivial flow graph, a single node with no edge; otherwise, the flow graph is *nonreducible* [All70, AlC76, CoS70].

Figure 5 shows the rest of the derived sequence for the example in Figure 4.

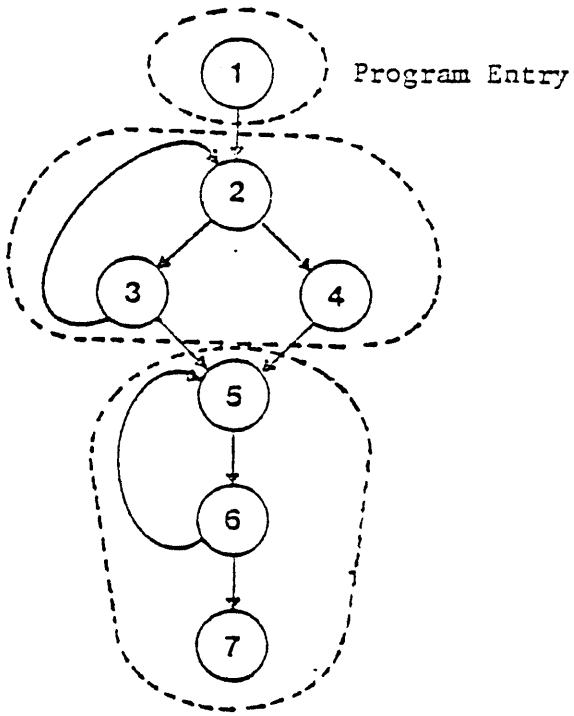


Figure 4. A flow graph with intervals.

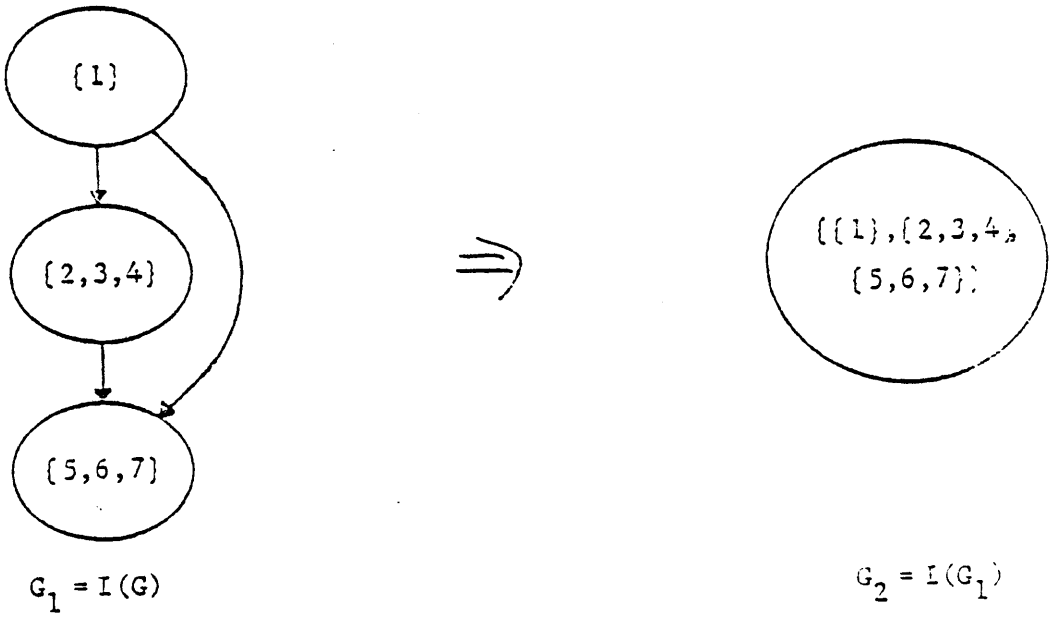


Figure 5. Derived sequence for Figure 4.

In this example, the graph is reducible; however, that will not always be the case, as Figure 6 demonstrates. If we apply Algorithm IP to this graph, The result will be the same graph -- each node is an interval unto itself.

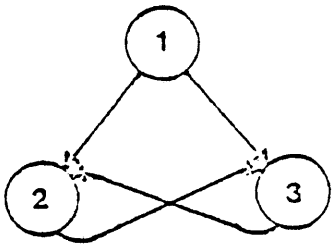


Figure 6. A nonreducible graph.

As it happens, the data flow analysis algorithms based on intervals work only for reducible graphs, so nonreducibility could present a serious obstacle. However, we are able to ignore this problem for two reasons. First, three empirical studies have shown that flow graphs arising from actual computer programs are almost always reducible -- i.e., more tha 95% of the time [AlC72, Knu71, KeZ77]. Second, any nonreducible graph can be transformed to a reducible one by a process known as *node splitting* [CoS70, Sch72]. Figure 7 shows a split version of the graph in Figure 6; the new graph, semantically identical to the old one, has been made reducible through the use of an exact copy of node 3.

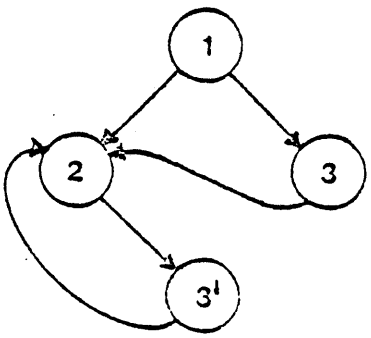


Figure 7. Split version of Figure 6.

Thus, secure in the knowledge that node splitting can always be applied in those rare cases where a graph fails to reduce, we can concentrate on finding fast data flow algorithms for reducible flow graphs.

Like all approaches which are based upon a program's control flow structure, the interval partition gives rise to a two-pass algorithm for data flow analysis. I will discuss the method as it applies to live analysis, treating each pass separately.

1) *Pass 1 -- Local to Global*

During the first pass, local quantities IN and THRU are computed for larger and larger regions of the program. The heart of this pass is Algorithm I1 below, which computes IN and THRU for an interval from their values for blocks in the interval. Note that a second parameter has been added to THRU to indicate a particular successor; this permits handling of THRU for composite regions like intervals.

Algorithm I1: Interval Pass 1.
Input:
1. An interval I.
2. IN(x), $\forall x \in I$; THRU(x,y), $\forall x \in I$, $\forall y \in S(x)$.

Output: IN(I); THRU(I,J), $\forall J \in S(I)$.
Auxiliary: For each $x \in I$, PATH(x), the set of variables A for which there is a clear path (not containing a store into A) from the entry of I to the entry of x.
Method:

begin
IN(I) := IN(h);
PATH(h) := Ω; ¢ Ω = set of all variables ¢
for all $x \in I-\{h\}$ in interval order **do**
PATH(x) := $\bigcup_{x \in P(x)}$(PATH(y) ∩ THRU(y,x));
IN(I) := IN(I) ∪ (PATH(x) ∩ IN(x))
od;
¢ let h_J denote the head of J ¢
for J such that $h_J \in S[I]$ **do**
THRU(I,J) := $\bigcup_{y \in P(h_J) \cap I}$(PATH($y$) ∩ THRU($y,h_J$))
od
end

If $G_0, G_1, \ldots, G_m$ is the derived sequence (where $G_0=G$), pass 1 consists of applying Algorithm I1 to each interval in G_0, then to each interval in G_1, and so on until it has been applied to the single interval in G_{m-1}. At this point, IN and THRU sets will have been computed for each node in the derived sequence of graphs.

2) *Pass 2 - Global to Local*

During the second pass, LIVE is computed for smaller and smaller regions of the program. Let x^* denote the single node in G_m. Pass 2 begins with the assignment

$$\text{LIVE}(x^*) := \text{IN}(x^*) .$$

This is clearly correct since x^* has no successors. The remainder of the pass consists of repeated application of Algorithm I2, which computes LIVE sets for each node in an interval I, given correct live sets for the entry to I and to each successor J of I. This precondition is assured by the order in which I2 is applied: first to the interval x^*, then to each interval in G_{m-2}, and so on (backwards through the derived sequence) until LIVE sets have been computed for every node in the original graph G.

The algorithm itself is based on the observation that if nodes of $I-\{h\}$ are treated in *reverse* interval order, the live analysis equation (**) can always be applied because the correct LIVE set for each successor of a given node $x \in I-\{h\}$ will have been previously computed. To see

this, suppose we are processing nodes of $I-\{h\}$ and we arrive at node x. A successor y of x can be one of three things:

1) y is another node in $I-\{h\}$, in which case LIVE(y) has already been computed because nodes are being treated in reverse interval order;
2) y is the head of I, in which case LIVE(I) can be used for LIVE(y), or
3) y is the head of some successor interval J, in which case LIVE(J) can be used.

Algorithm I2 is a direct encoding of these insights.

Algorithm I2: Interval Pass 2.
Input:
1. An interval I with head h.
2. IN(x), $\forall x \in I$; THRU(x,y), $\forall x \in I$, $\forall y \in S(x)$.
3. LIVE(I); LIVE(J), $\forall J \in S(I)$.

Output: LIVE(x), $\forall x \in I$.
Method:

begin
LIVE(h) := LIVE(I);
for all $J \in S(I)$ **do**
LIVE(head of J) := LIVE(J);
od;
for all $x \in I-\{h\}$ in reverse interval order **do**

$$\text{LIVE}(x) := \text{IN}(x) \cup \bigcup_{y \in S(x)} (\text{THRU}(x,y) \cap \text{LIVE}(y));$$

od
end

Although interval analysis has been shown to require fewer bit vector operations than the iterative method in many cases [Ken76], it is still $O(n^2)$ in the worst case, and in practical implementations the elegantly simple iterative method may prove faster. The main advantage of the interval approach is that it constructs a representation of the program control flow structure which can be used for other optimizations [Coc70]. Allen, Cocke, Schwartz, Kennedy, Aho and Ullman [All70, Coc70, AlC76, CoS70, Ken71, Ken76, AhU73] have applied interval analysis in the solution of data flow problems. Allen and Cocke [All70, Coc70] first used intervals to solve class 1 (forward) problems, while Kennedy [Ken71, Ken76] indicated the interval solution for class 2 (backward) problems.

3.4 T1-T2 Analysis

In search of better theoretical results and faster algorithms, Ullman [Ull73] introduced two transformations on program graphs. Transformation T1 collapses a self-loop to a single node, while transformation T2 collapses a sequence of two nodes to a single node if the second has the first as its only predecessor. When T1 and T2 are repeatedly applied to a control flow graph, the graph is often reduced to a single node. Hecht and Ullman [HeU72] have shown that the reducible flow graphs in the T1-T2 sense are exactly the interval-reducible graphs. This result has led to a number of useful characterizations of flow graph reducibility [HeU72, HeU74].

T1-T2 analysis also allowed Ullman [Ull73] to design an algorithm which uses balanced "3-2" trees to perform available expression computation in O(n log n) extended steps. Ullman's method can be extended to many other class 1 problems; however it is not known whether it can be adapted to class 2 problems.

3.5 Node Listings

A variation of the iterative method for data flow analysis builds an intermediate representation of the control flow called a *node listing* [Ken75a], which is then used to solve the data flow equations. I here describe the node listing method for live analysis.

In the solution of the live analysis problem we are concerned with how operations in one block can effect "liveness" on entry to another. Thus we are interested in propagating information from every block in the program to every other block. Thus it is natural to consider the paths along which this information is propagated. A *node listing* for control flow graph $G=(N,E,n_0)$ is defined to be a sequence

$$\ell=(n_1,n_2,\ldots,n_m)$$

of nodes from N (nodes may be repeated) such that every simple path in G is a subsequence of ℓ. That is, if

$$(x_1,x_2,\ldots,x_k)$$

is a simple path in G then there exist indices

$$j_1,j_2,\ldots,j_k$$

such that $j_i<j_{i+1}$, $1\le i<k$, and $x_i = n_{j_i}$, $1\le i\le k$.

For any control flow graph there exists a node listing of length $\le n^2$ where $n=|N|$ since

$$\ell = (n_1,n_2,\ldots,n_n,n_1,n_2,\ldots n_n,\ldots,n_1\ldots,n_n)$$

with n repetitions of $(n_1,\ldots,n_n)$ is certainly such a listing. A node listing is *minimal* if there is no shorter listing for G.

The utility of this concept is demonstrated by the following algorithm which, given a node listing, computes the live sets in a manner similar to the Hecht-Ullman iterative method.

Algorithm NL: Node Listing Live Analysis.
Input: IN(b), THRU(b), $\forall b\in N$.
Output: LIVE(b), $\forall b\in N$.
Method:

```
begin
   for all b∈N do LIVE(b) := IN(b) od;
   for i := |nodelist| to 1 by -1 do
      b := nodelist[i];
      LIVE(b) := IN(b) ∪ ⋃ (THRU(b) ∩ LIVE(x))
                        x∈S(b)
   od
end
```

The node listing concept is introduced in [Ken75a]; in [AhU75] Aho and Ullman show that for reducible flow graphs on O(n log n) length node listing can be found in O(n log n) time. Combining this method with Algorithm NL produces an O(n log n) algorithm to solve either class 1 or class 2 data flow problems. Markowsky and Tarjan [MaT75] have shown that O(n log n) is a lower bound of the node listing algorithm, i.e., no better worst-case bound can be found, although there are linear listings for a large class of graphs [Ken75]

3.6 Path Compression

Another O(n log n) data flow analysis algorithm was discovered by Graham and Wegman [GrW76]. It is based on three transformations which are similar to Ullman's T1 and T2. The Graham-Wegman transformations are depicted in Figure 8. Transformation T_1 removes a self loop; T_2 compresses a two-step path to a one-step path, eliminating the middle node whenever it has no other successors (T_2b); T_3 eliminates a successor of the entry node that has no successors of its own. For technical reasons, application of T_1 requires that the node with the loop have a unique predecessor. An example reduction using these transformations is shown in Figure 9. Graham and Wegman have shown that any graph reducible in the interval sense will be reduced by T_1-T_3.

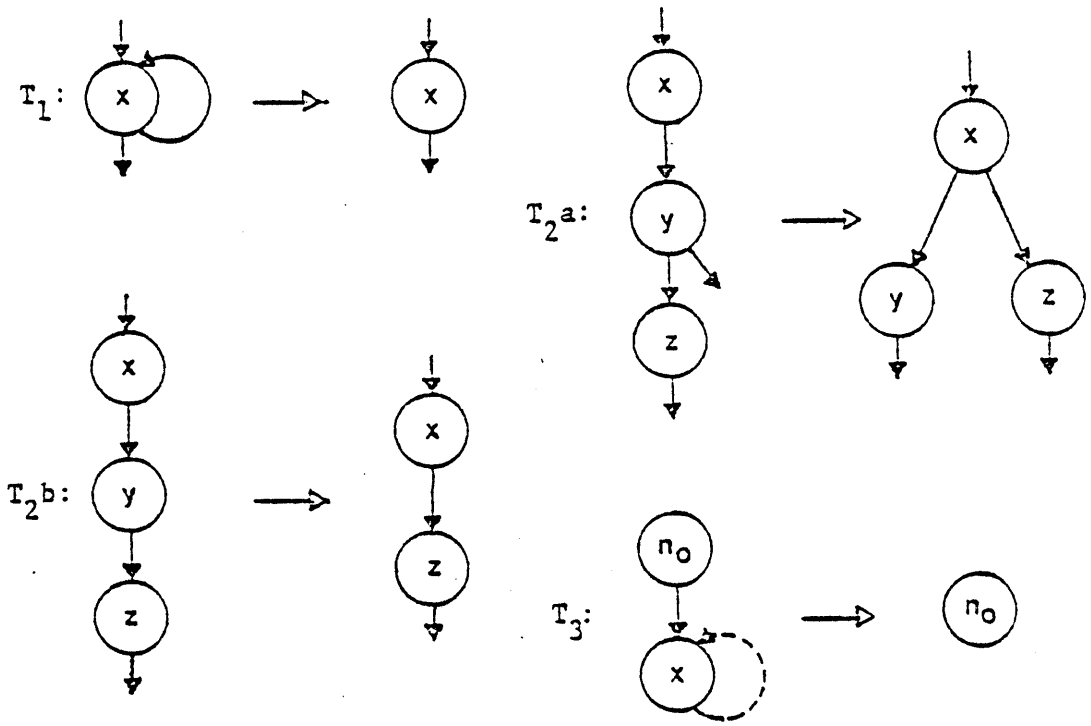


Figure 8. Graham-Wegman path compression transformations.

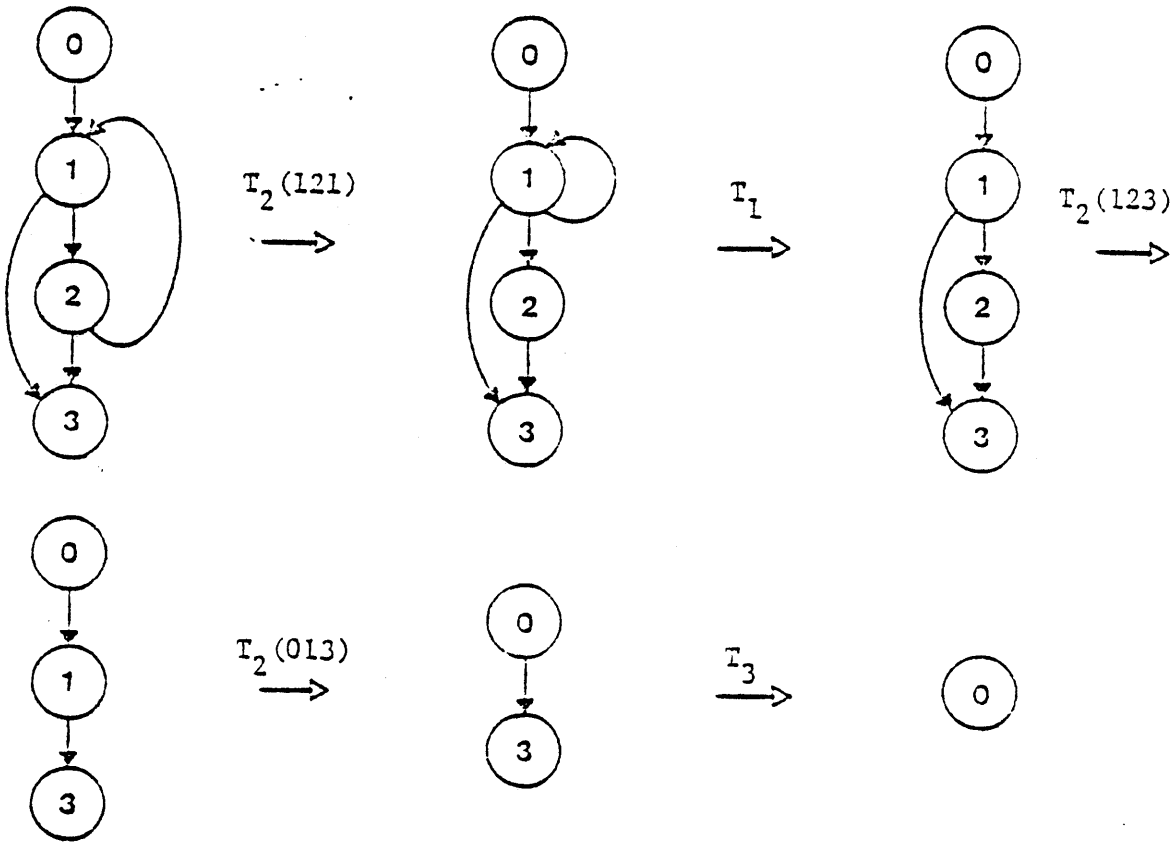


Figure 9. Sample Graham-Wegman reduction.

Data flow analysis using the path compression transformations is similar to interval analysis. The method I present here differs from the one originally published by Graham and Wegman in that it easily handles backward as well as forward analysis.

Given a flow graph, the first step is to construct a "parse". i.e., a list of transformations which will reduce the graph to a single node. The complexity analysis is very sensitive to the order to choose a parse that reduces loops from the inside out and minimizes the number of T_2 transformations. Since T_2 transformations are the most expensive, this strategy achieves the good time bound.

Once available, the parse is employed in a two-pass algorithm which computes IN and THRU for composite regions of increasing size in a pass through the reduction sequence, then computes LIVE for each node as it appears in the reverse reduction sequence (or *production* sequence). This process is embodied in Algorithm P2, which applies a set of associated computations at each reduction or production. Each transformation in the parse is really a pair $<t, \eta>$, where t is a transformation number and η is a mapping from the nodes in the production to nodes of the graph being reduced; in other words, η specifies the region of application for transformation t. Such a pair is called a *transformation instance.*

Algorithm P2: Two-pass Live Flow Analysis

Input:

1. A graph $G=(N,E,n_0)$.
2. IN(x), $\forall x \in N$; THRU(x,y), $\forall x \in N$, $\forall y \in S(x)$.
3. A list PARSE, consisting of transformation instances $<t,\eta>$ which reduce G.

Output: LIVE(x), $\forall x \in N$.

Method:

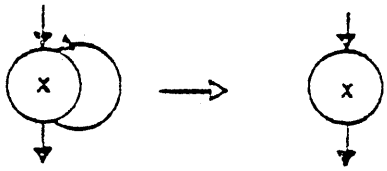
```
begin
   ¢ pass 1 ¢
   for i := 1 to |PARSE| do
      <t,η> := PARSE[i];
      apply the reduction computations  associated with t to the nodes specified by η.
      od;
   LIVE(n0) := IN(n0);
   ¢ pass 2 ¢
   for i:= |PARSE| to 1 by −1 do
      <t,η> := PARSE[i];
      apply the production computations  associated with t to the nodes specified by η.
      od
   end
```

All that remains is to specify the computations associated with each transformation. Figure 10 shows the computations of IN and THRU performed during the reduction pass. Note that path compression emphasizes edges rather than nodes, so the THRU sets being constructed are for composite edges. For notational convenience, we define THRU of a nonexistent edge to be the empty set. Figure 11 shows the production computations; an initial LIVE set for each node is determined when the node first appears as the result of some production. This live set is then revised as new exit edges are added by T_2a productions.

In practice, path compression is very fast indeed; in fact, it operates in linear time for an extremely large subclass of the reducible flow graphs. Its only disadvantage is that, although classified as a "structured" method, the structure it uncovers seems unnatural because it is based on edges rather than nodes. Nevertheless, path compression is an excellent algorithm from both the theoretical and practical standpoints.

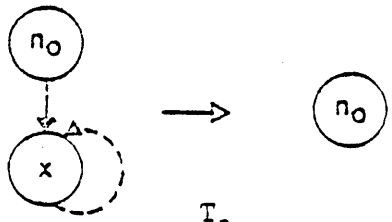
3.7 Balanced Path Compression

In 1975, Tarjan devised an algorithm [Tar75b] which combined elements of the node listing approach with a stronger form of path compression using a balanced tree data structure he had introduced in [Tar75a]. The result is a very fast algorithm with running time $O(n\alpha(n,n))$, where α is related to a functional inverse of Ackermann's function. Thus for all practical purposes the algorithm is asymptotically linear; unfortunately it seems very complex, so until there is some experience with an implementation, I cannot tell whether it is suitable for inclusion in a compiler. Tarjan's algorithm can be used to solve a variety of class 1 problems, but it is not yet clear that it can be adapted to class 2 problems.



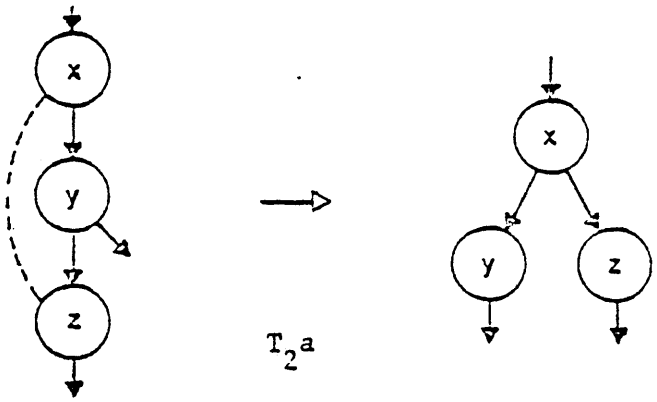
T_1

no computation



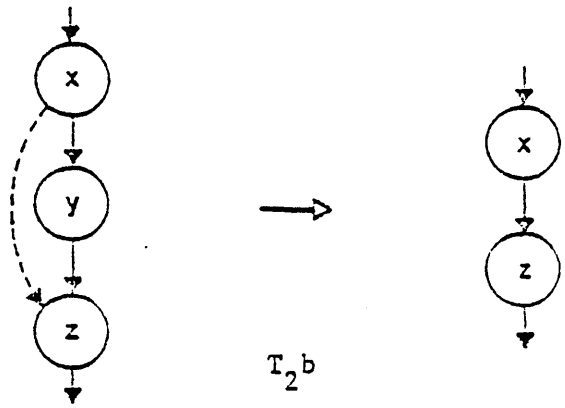
T_3

$IN(n_o) := IN(n_o) \cup (THRU(n_o,x) \cap IN(x))$



$T_2{}^a$

$THRU(x,z) := THRU(x,z) \cup (THRU(x,y) \cap THRU(y,z))$

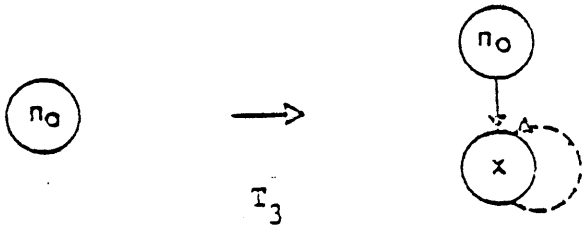


$T_2{}^b$

$IN(x) := IN(x) \cup (THRU(x,y) \cap IN(y))$

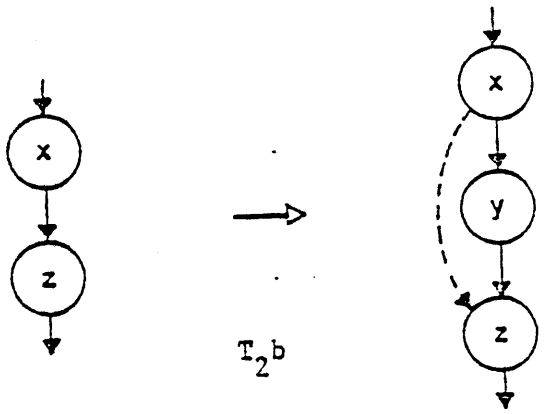
$THRU(x,z) := THRU(x,z) \cup (THRU(x,y) \cap THRU(y,z))$

Figure 10. Reduction computations.



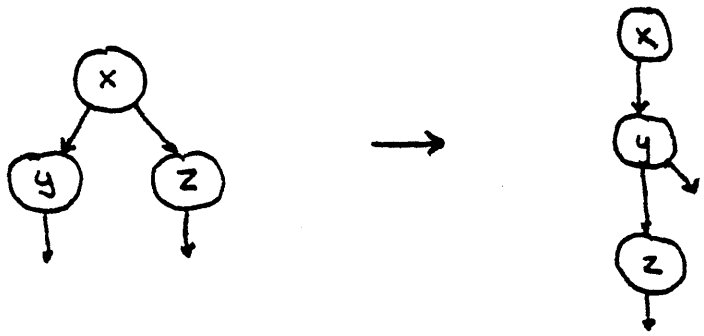
T_3

LIVE(x) := IN(x)



T_2b

LIVE(y) := IN(y)∪(THRU(y,z)∩LIVE(z))



LIVE(y) = LIVE(y) ∪ (THRU(y,z) ∩ LIVE(z))

Figure 11. Production computations.

3.8 Graph Grammars

in an attempt to further simplify the problem of data flow analysis, Farrow, Kennedy and Zucconi [FKZ75] studied further restrictions on the class of acceptable graphs, restrictions stronger than the traditional notion of reducibility. They introduced the *Semi-Structured Flow Graph* (SSFG) grammar, depicted informally in Figure 12, and studied the class of flow graphs generated by that grammar. The set of rules in Figure 12 was chosen because it seems to include most of the control structures proposed as extensions of the basic Böhm and Jacopini set for structured programming [BoJ66]. For example, the SSFG grammar can generate the double-exit loop used by Ashcroft and Manna [AsM71] to demonstrate a limitation of the Böhm-Jacopini control structures (see Figure 13).

The major problem with using SSFG or any other graph grammar for data flow analysis is that of *graph parsing*, constructing a parse for an arbitrary graph. For the SSFG rules, an important step toward the fast parsing algorithm was a proof that corresponding SSFG reductions can be applied in any order without affecting the result. In other words, reducibility of a given graph is not sensitive to the order in which reductions are applied. Farrow, Kennedy and Zucconi established this result by proving, via a long graphical argument, that the SSFG reductions have the *Finite Church-Rosser* property [ASU72, Set74]. As a result of this property, they were able to devise a parsing algorithm which applies reductions in a disciplined way and avoids wandering around the graph.

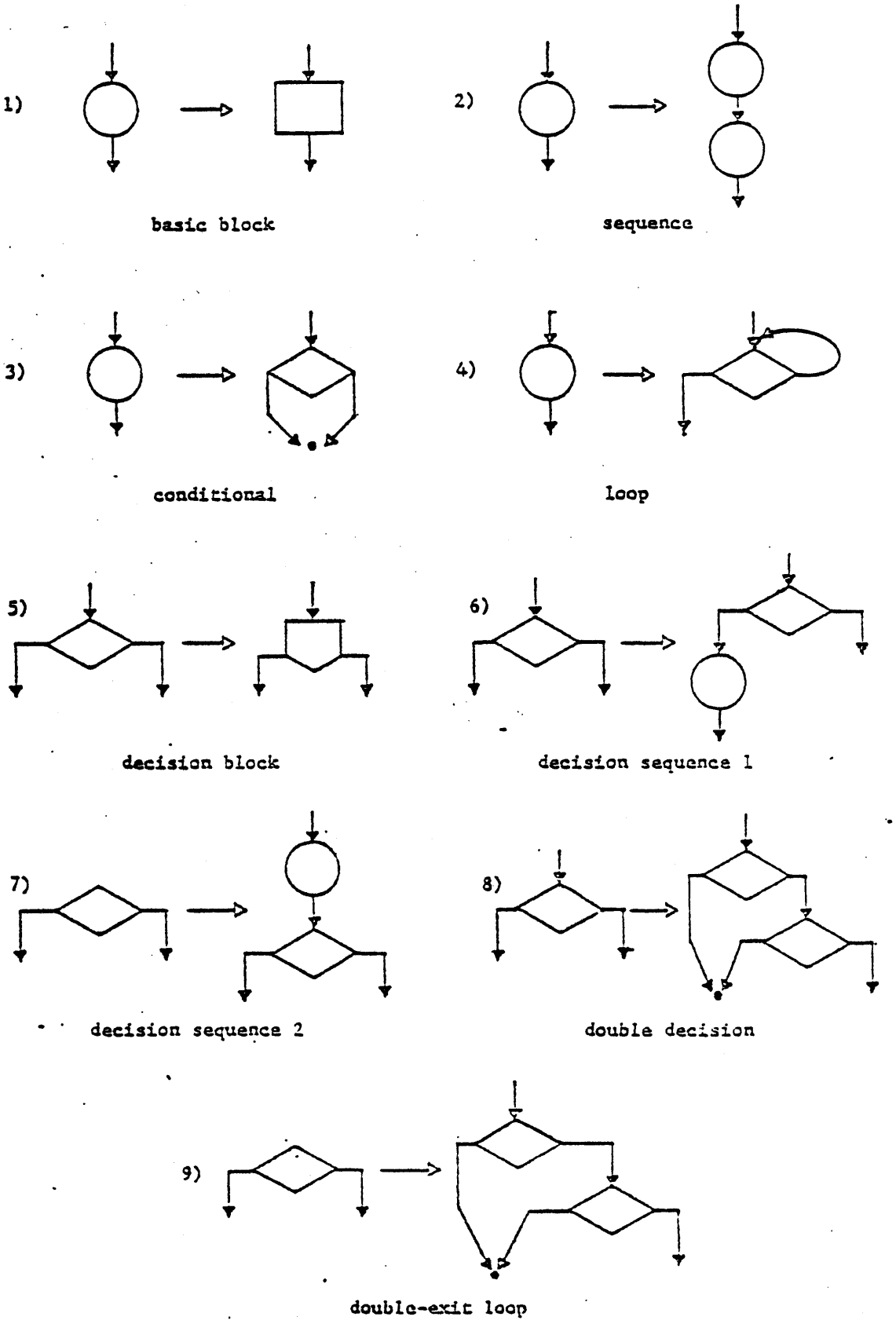


Figure 12. SSFG grammar.

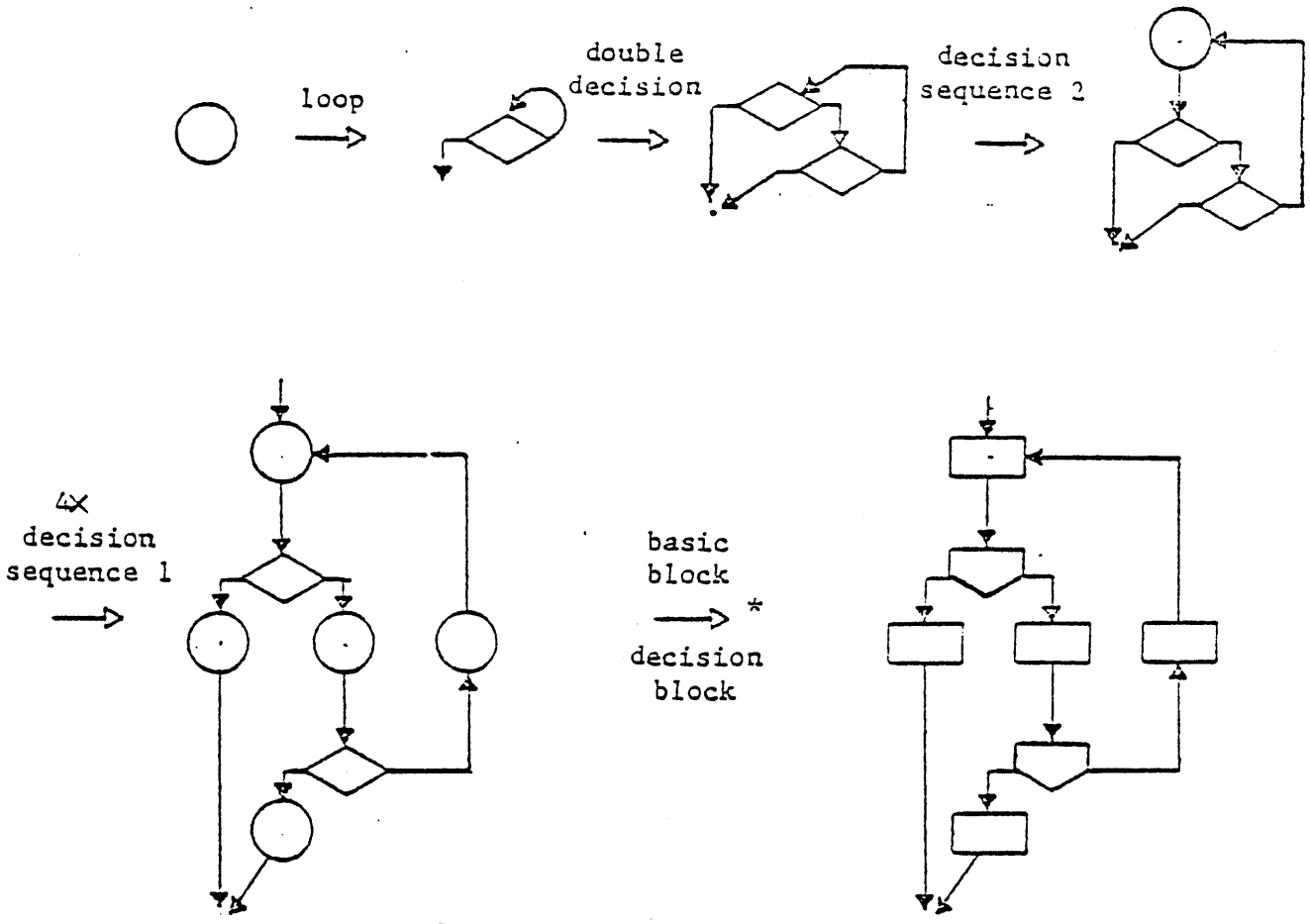


Figure 13. Derivation of the Ashcroft-Manna counterexample.

I present the parsing algorithm in two parts. First, Algorithm CO (collapse) finds all the reductions which apply at a particular node x. If it discovers at least one reduction, it sets a success flag to **true** and returns the reduction list.

Algorithm CO: Collapse
Input: A graph Γ and a node x in Γ.
Output:
1. A flag *SUCCESS* indicating whether or not a reduction has been found,
2. A list of reductions P_x (possibly empty),
3. A modified graph Γ'.

Method:

begin $P_x := \epsilon$; *SUCCESS* := **false**;
  reducing := **true**; $\Gamma' := \Gamma$;
  while *reducing* **do**
    for each production P in G_{SSFG} **do**
      if *right-hand-side*(P) is isomorphic to a region R in Γ' headed by x
      then
        apply P^{-1} to reduce R to a single node x', forming a new version of Γ';
        add the production P to P_x along with some auxiliary information;
        $x := x'$;
        SUCCESS := **true**;
        goto *reduced*
        fi

```
        od;
      reducing := false;
    reduced:
      skip
      od
  end
```

The SSFG parsing algorithm assumes a list L of nodes of the program in *straight order*, a fairly obvious order for nodes of the flow graph [EBA72, HeU75], and produces a parse P_Γ. The basic scheme is to take each node from L in sequence and try a collapse. Whenever a collapse succeeds, the algorithm backs up to a predecessor, indicated by a "link," to try further collapses; otherwise it moves on to the next node on L. This disciplined backup is the key to a linear time bound.

Algorithm PA: SSFG Parse

Input:

1. A graph Γ.
2. A list L of nodes of Γ in straight order.

Output:

1. A list P_Γ of reductions.
2. An answer to the question, "is Γ in the language generated by G_{SSFG}?"

Method:

```
begin
  L := the list of unvisited nodes (straight order);
  x := the entry of Γ;
  P_Γ := ϵ;
  remove x from L;
  while x ≠ null do
    perform a collapse at node x;
    ¢ collapse produces Γ', P_x, and the flag SUCCESS ¢
    make x the unique linked predecessor of all unvisited successors of x in Γ';
    append P_x to P_Γ;
    Γ := Γ';
    if SUCCESS ¢ at least one reduction ¢
          and x is linked to a predecessor
      then x := linked predecessor of x
      elif L=ϵ then x := null
      else x := hd L; L := tl L
      fi
    ;
  if Γ is now a single computation node
  then the graph is SSFG and P_x is a valid parse
  else the graph is not SSFG fi
  end
```

The operation of this algorithm is demonstrated by the example in Figure 14. in this figure, links are indicated by dotted lines. Nodes are numbered in straight order. The steps are as follows:

1) An unsuccessful collapse is attempted at node 1. A link to 1 is inserted in 2.
2) A collapse at node 2 discovers a "decision sequence 1" involving node 4. Links to 2 are inserted in nodes 3 and 10 (Figure 14b).
3) A backup leads to another unsuccessful collapse at 1.

4) A collapse at node 3 discovers a long sequence of reductions:
   two "decision sequence 1" reductions (Figure 14c), a "double-exit loop" and a "decision sequence 1" (Figure 14d), a "conditional" and a "decision sequence 2" (Figure 14e). A link to 3 is inserted in 10, but *not* in 2 (it has been visited).
5) After a backup, a collapse at node 2 discovers a "double-exit loop", a "conditional" and a "sequence" (Figure 14f).
6) After one more backup, a collapse at node 1 produces the final "sequence" reduction.

It has been shown that this algorithm, in time linear in the number of blocks in the original program, either produces a parse for Γ or reports that Γ is not reducible. If the graph is reducible, the length of its parse must also be linear in the size of the original graph.

With the parse in hand, we can apply the same two-pass algorithm used by path compression (Algorithm P2) to perform data flow analysis. Space does not permit me to specify the computations associated with each of the nine transformations in the SSFG grammar; instead, I have selected two rules, "sequence" and "double-exit loop", as examples. Reduction computations for these rules are shown in Figure 15 and production computations in Figure 16. As with path compression, a correct LIVE set is determined for each node when it first appears as the result of some production. Since there is a fixed number of operations associated with each transformation in the parse, the linear parse length implies that the entire computation takes linear time.

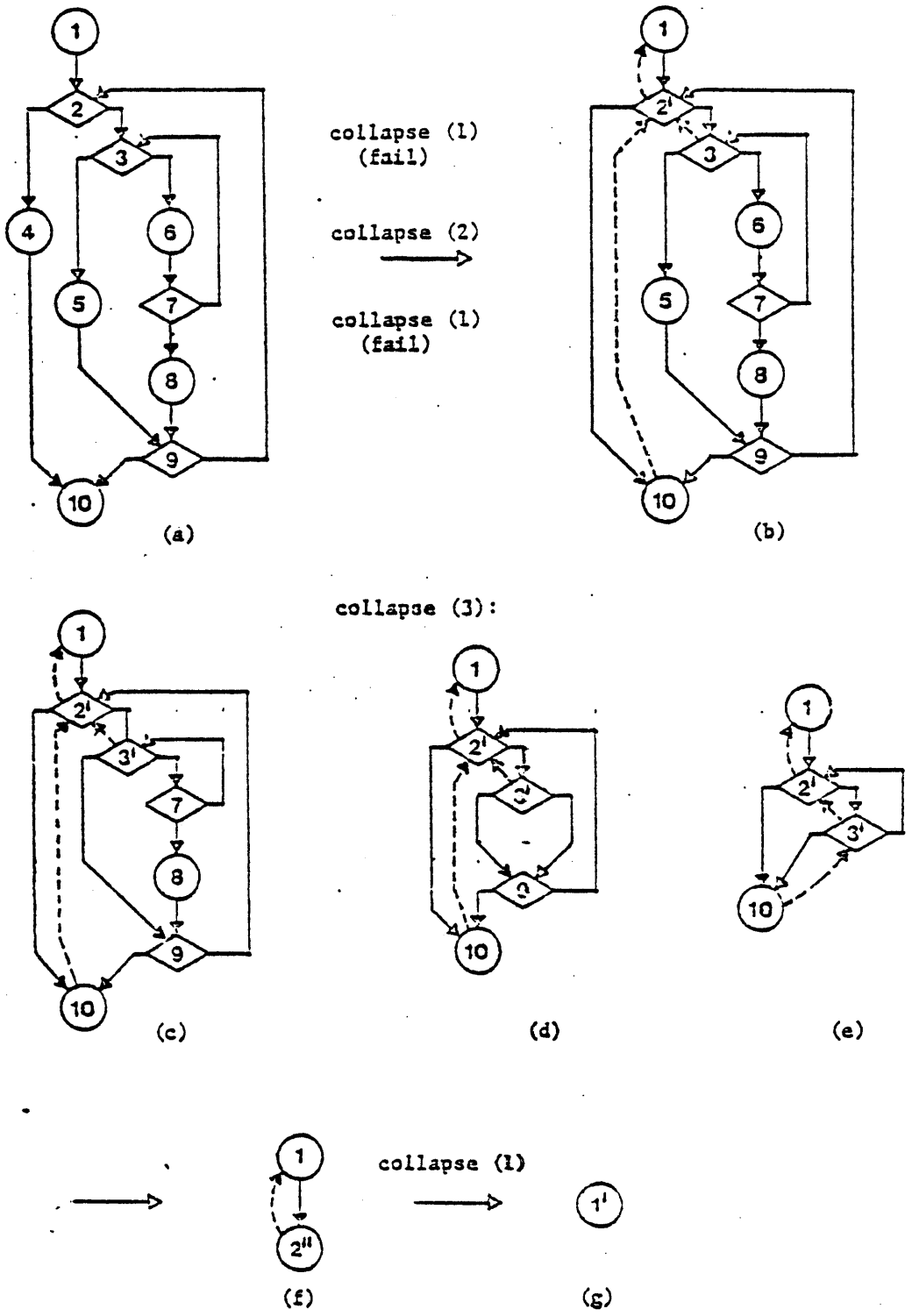
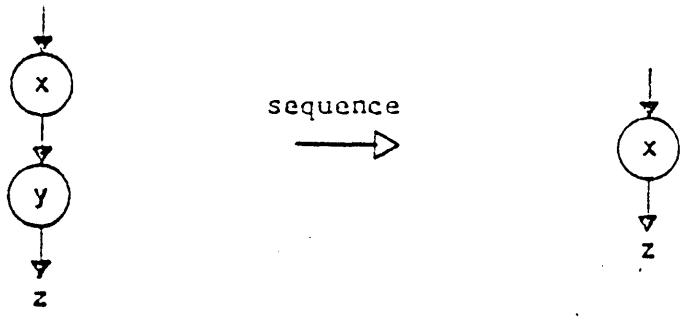


Figure 14. An example parse.

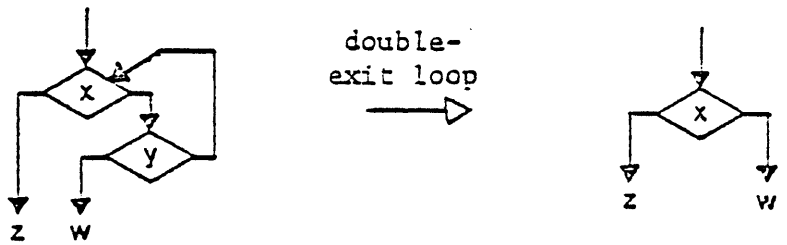
2)



$IN(x) := IN(x) \cup (THRU(x,y) \cap IN(y))$

$THRU(x,z) := THRU(x,y) \cap THRU(y,z)$

9)



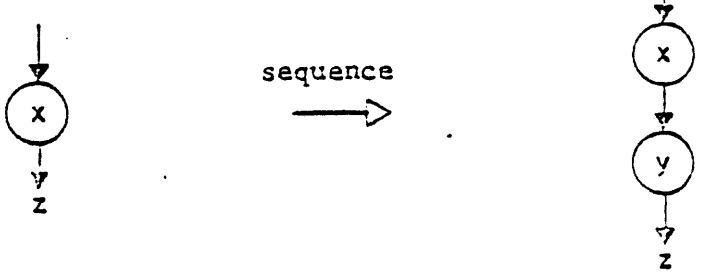
$IN(x) := IN(x) \cup (THRU(x,y) \cap IN(y))$

$THRU(x,z) := THRU(x,z)$

$THRU(x,w) := THRU(x,y) \cap THRU(y,w)$

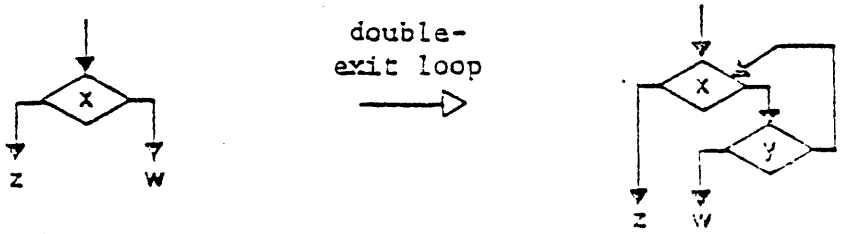
Figure 15. Sample reduction computations.

2)



$LIVE(y) := IN(y) \cup (THRU(y,z) \cap LIVE(z))$

9)



$LIVE(y) := IN(y) \cup (THRU(y,x) \cap LIVE(x))$

$\cup (THRU(y,w) \cap LIVE(w))$

Figure 16. Sample production computations.

An important by-product of the method is the parse itself, which can be used for many different data flow problems and which provides a convenient representation of the structure of the program. Because it uncovers loops and other control constructs this representation can be used to perform optimizations like code motion and strength reduction. The structure discovered by the SSFG parse is more natural than that discovered by the interval method or the Graham-Wegman technique, because the SSFG grammar is based upon control structures arising from good programming practice.

The main drawback of the graph grammar approach is its limited range of applicability. In order to find out how much of a drawback that is, Kennedy and Zucconi conducted a follow-up study in which they analyzed 500 FORTRAN subroutines taken from running programs used by several departments in the School of Natural Sciences at Rice University. All of these programs were written before the emphasis on structured programming, yet 94% were Cocke-Allen reducible and, of these, 88% were SSFG reducible. In other words, 88% of the programs for which most other methods work can be reduced and hence analyzed by the SSFG method [KeZ77].

As a final note I would point out that the Graham-Wegman algorithm is also linear on all the SSFG-reducible graphs. It is gratifying to observe that well-structured programs can produce benefits other than the obvious ones -- e.g., faster compilation speeds. In a sense, programs that are easier for humans to understand are also easier for compilers to understand.

3.9 High-Level Data Flow Analysis

The methods surveyed thus far are designed to work with a low-level version of the program. One might well ask if it is possible to perform the same analysis on a high-level representation such as the parse tree. The answer is yes. This approach, often called *high-level data flow analysis,* is similar to the graph grammar method, except no complicated graph parsing algorithm is required. For simplicity, I will illustrate the method by considering a language which contains no *escape* or *goto* statements. Consider the simple grammar fragment below.

<program>::= **begin** <statement> **end**
<statement>::=<assignment>
<statement>::=<statement> ; <statement>
<statement>::=**if** <condition> **then** <statement> **else** <statement> **fi**
<statement>::=**while** <condition> **do** <statement> **od**

Although this grammar is clearly ambiguous, we can nevertheless write a parser which resolves the ambiguity in some sensible way, say by grouping from left to right.

The parse tree for a program generated by this grammar will have a *<program>* node as its root and a number of *<statement>* nodes as nonterminals in the tree. Data flow analysis can be applied to such a tree in the familiar two-pass fashion. The first pass propagates IN and THRU sets associated with *<statement>* nonterminals up toward the root; the second pass propagates LIVE sets down toward the leaves. To specify the entire procedure within this framework, one need only specify the computations that can occur at each *<statement>* node — for pass 1, how to compute IN and THRU for a *<statement>* given IN and THRU for its parts, and for pass 2, how to compute LIVE for subparts of a *<statement>* given LIVE for the *<statement>* along with IN and THRU for the parts, as determined on pass 1. These specifications must be given for each rule of the grammar.

As an illustration, consider the computations associated with the sample grammar given earlier. For compactness, I will specify these computations using the shorthand notations *S* for *<statement>*, *C* for *<condition>*, *P* for *<program>*, and *A* for *<assignment>*; I will use

subscripts to distinguish different occurrences of the same nonterminal in a single rule. Each nonterminal S will have a number of associated *attributes*: IN, THRU, LIVE, and LIVEOUT (the set of variables live on exit) for the region that S represents. The specification is completed by associating with each rule of the grammar *semantic equations*, which show how to compute the various attributes. To apply the semantic equations at a particular node while traversing the parse tree, set up a correspondence between the node and its sons on the one hand and the nonterminals of the production that applies at the node on the other. Then the semantic equations associated with the rule can be used to compute attributes for the tree nodes.

Here is the complete specification for the sample grammar.

1) P := **begin** S **end**
   ¢ no computations on pass 1 ¢
   ¢ pass 2 computations ¢
   LIVE(S) := IN(S);
   LIVEOUT(S) := ϕ ;

2) S ::= A

   ¢ pass 1 ¢
   IN(S) := IN(A);
   THRU(S) := THRU(A);
   ¢ pass 2 ¢
   LIVE(A) := IN(A) ∪ (THRU(A)∩LIVEOUT(S));

2) S_0 ::= S_1 ; S_2

   ¢ pass 1 ¢
   IN(S_0) := IN(S_1) ∪ (THRU(S_1)∩IN(S_2));
   THRU(S_0) := THRU(S_1) ∩ THRU(S_2);
   ¢ pass 2 ¢
   LIVEOUT(S_2) := LIVEOUT(S_0);
   LIVE(S_2) := IN(S_2) ∪ (THRU(S_2)∩LIVEOUT(S_2));
   LIVEOUT(S_1) := LIVE(S_2);
   LIVE(S_1) := IN(S_1) ∪ (THRU(S_1)∩LIVEOUT(S_1));

4) S_0 ::= **if** C **then** S_1 **else** S_2 **fi**

   ¢ pass 1 ¢
   IN(S_0) := IN(C) ∪ (THRU(C)∩(IN(S_1)∪IN(S_2)));
   THRU(S_0) := THRU(C) ∩ (THRU(S_1)∪THRU(S_2));
   ¢ pass 2 ¢
   LIVEOUT(S_1) := LIVEOUT(S_2) := LIVEOUT(S_0);
   LIVE(S_1) := IN(S_1) ∪ (THRU(S_1)∩LIVEOUT(S_1));
   LIVE(S_2) := IN(S_2) ∪ (THRU(S_2)∩LIVEOUT(S_2));
   LIVEOUT(C) := LIVE(S_1) ∪ LIVE(S_2);
   LIVE(C) := IN(C) ∪ (THRU(C)∩LIVEOUT(C));

5) S_0 ::= **while** C **do** S_1 **od**

   ¢ pass 1 ¢
   IN(S_0) := IN(C) ∪ (THRU(C)∩IN(S_1));
   THRU(S_0) := THRU(C);

¢ pass 2 ¢
LIVEOUT(C) := LIVEOUT(S_0) ∪ IN(S_1) ∪ (THRU(S_1)∩IN(C));
LIVE(C) := IN(C) ∪ (THRU(C)∩IN(C));
LIVEOUT(S_1) := LIVE(C);
LIVE(S_1) := IN(S_1) ∪ (THRU(S_1)∩LIVEOUT(S_1));

The high-level approach, described here via an *attributed grammar* [Knu68], has several advantages. First, because the computations at each node of the parse tree are selected from a finite set and because the tree is traversed exactly twice, the total amount of processing is linear in the number of nodes of the parse tree. However, the constant of proportionality depends on the richness of the set of control structures — the richer the language, the more complex the data flow analysis.

Second, the method lends itself to convenient updating of data flow when sections of the parse tree are modified by optimization. If the leaf of some subtree is changed, new values of IN and THRU can be propagated upward to the first nonterminal where these sets are unchanged; then the computation of modified LIVE sets can be propagated back toward the leaves. This process limits the updating in response to a change to the region where the change actually makes a difference.

Finally, the first pass of high-level analysis can be performed as a part of the parse itself. Whenever a composite control structure is recognized, the IN and THRU sets for the region it represents are computed from IN and THRU for the individual parts according to the semantic equations above.

Various formulations of high-level data flow analysis have been proposed [Wul75, NeA75, Jaz75]. Particularly notable is its use in the BLISS/11 compiler at Carnegie-Mellon [Wul75]. The name "high-level data flow analysis" was coined by Rosen in his detailed treatment of the method [Ros77]. Recent work [BaJ78a, BaJ78b, Ros77, Ros79] in high-level analysis allows the same *escape* and *goto* statements allowed by low-level analysis. In most cases, such jumps can be processed without a substantial increase in computational complexity.

3.10 Summary Table

Table 4 summarizes the characteristics of the algorithms I have described. The column labeled *speed* shows the asymptotic complexity of each method. In the *simple* column, "S" indicates an easy-to-program method, "C" indicates a complicated method, and "M" indicates average difficulty. A "yes" under *structure* says that the method uses a model of the program loop structure in its computation -- i.e., that the algorithm attempts to discover the structure of the program.

Method	*speed*	*simple?*	*structure?*	*both ways?*	*graph class*
Iterative	n^2	S	no	yes	all
Interval	n^2	M	yes	yes	reducible
Bal. Tree	n log n	C	yes	no	reducible
Path Comp.	n log n	M	semi	yes	reducible
Node List	n log n	M	no	yes	reducible
Bal. Path	$n\alpha(n,n)$	C	no	?	reducible
Grammar	n	M	yes	yes	L(grammar)
High-Level	n	S	yes	yes	parse trees

Table 4. Summary of data flow methods

A "yes" in the *both ways* column indicated that the algorithm works in the given time on both forward and backward data flow problems. The last column shows the class of graphs for which each algorithm was analyzed (in most cases this is also the class to which the algorithm is applicable).

3.11 Interprocedural Analysis

The foregoing material has said nothing about the effect of procedure calls on data flow analysis. Usually calls within blocks are treated as complex instructions which may affect the values of many variables. It is the function of *interprocedural data flow analysis* [All74] to construct *summary information* for a procedure: which variables are used and which are redefined as the result of a call. For example, interprocedural analysis might construct IN and THRU sets for the procedure call to support live analysis.

Interprocedural analysis is important because, in its absence, extremely conservative assumptions must be made. For example, in live analysis, it must be assumed that a procedure uses every variable it has access to; in availability analysis it must be assumed that it kills every expression it can and defines no new ones. Broad assumptions like these quickly dilute the power of data flow analysis.

Interprocedural analysis is a complex process, particularly for languages with Algol-like scoping rules. One method [Bar78] entails constructing a call graph and summary information for a single activation of each procedure in the graph, then taking a transitive closure on the graph. Another approach is to adapt intraprocedural methods like the ones described earlier in this section to interprocedural use, applying them to the call graph or within the procedures themselves [Ban79, Ros79]. Since it is treated elsewhere in this collection, I will not discuss it in detail, but be aware that interprocedural analysis is an essential part of any system for global data flow analysis.

4. USE-DEFINITION CHAINS

For data flow analysis problems which are more complex than the ones examined previously, data interconnections may be expressed in a pure form which directly links instructions that produce values to instructions that use them. These links are called *use-definition chains* For the purposed of this exposition, I will assume that these chains are realized in the following forms:

1) For each instruction i and input variable V, DEFS(V,i) is the set of instructions which may be the most recent defining instructions for V at runtime. In other words, DEFS(V,i) contains the set of instructions which may compute the value of V used by i.

2) For each instruction i and output variable V, USES(V,i) is the set of instructions which may use the value of V computed by i at runtime. These sets are related as follows:

$$x \in \text{DEFS}(A,y) \equiv y \in \text{USES}(A,x).$$

I will postpone, for the moment, a discussion of how use-definition chains are used in favor of a discussion of how to compute the sets DEFS and USES. Suppose we are considering an instruction y and an input variable A. If there is a defining instruction x earlier in the same block, then this is the only possible member of DEFS(A,y). Otherwise, we must discover which instructions in the program compute values that can "reach" the beginning of the block; every such instruction that has A as its output variable should be in DEFS(A,y). Thus the

problem is reduced to computing, for each block b in the program, the set REACHES(b) of pointers to instructions that compute values which are available on entry to b. Let DEFOUT(y,x) be the set of instructions in block y which produce values that are still available on entry to successor x, and let NKILL(y,x) be the set of instructions whose output variables are not redefined in passing through block y to block x. Then the following system of equations holds.

$$\text{REACHES}(n_0) = \phi$$

(***)

$$\text{REACHES}(x) = \bigcup_{y \in P(x)} (\text{DEFOUT}(y,x) \cup (\text{REACHES}(y) \cap \text{NKILL}(y,x)))$$

This is exactly the kind of system which can be solved by any of the data flow analysis methods described in Section 3.

Once DEFS is available, USES can be produced by simple inversion. The informal algorithm below can be used for this purpose.

Algorithm US: USES Computation
Input: DEFS.
Output: USES.
Method:

```
begin
   USES(*) := φ;
   for each instruction i in the program do
      for each input variable A of instruction i do
         for each instruction j in DEFS(A,i) do
            USES(output(j),j) := USES(output(j),j) ∪ {i}
         od
      od
   od
end
```

To illustrate the usefulness of these chains, I present an application to dead code elimination. The usual method for eliminating dead code is to first find and mark all instructions which are "useful" in some sense. This is done by starting with a set of *critical instructions*, instructions which are useful by definition. For example, you might declare all output instructions to be critical. Once every instruction in the critical set is marked, the method proceeds to mark any instruction that defines a variable used by at least one marked instruction, continuing until no more instructions can be marked. The use-definition chains help in the location of instructions which can compute some input of a marked instruction. To manage the process, Algorithm MK below uses a workpile of instructions ready to be marked.

Algorithm MK: Mark Useful Instructions
Input:
1. Use-def chains, DEFS(v,i).
2. Set of critical instructions CRIT.

Output: For each instruction i, MARK(i) = **true** iff i is useful.
Method:

```
begin
   MARK(*) := false;
   PILE := CRIT;
   while PILE ≠ φ do
      x := an arbitrary element of PILE;
      PILE := PILE − {x};
      MARK(x) := true;
      for each y ∈ DEFS(A,x) do
         if ¬MARK(y) then
         PILE := PILE ∪ {y}
            fi
         od
      od
   end
```

All that remains after application of the marking algorithm is to remove any unmarked instructions as useless.

While Algorithm MK demonstrates a fairly powerful application of use definition chains, it only uses chains in one direction. We shall next consider the problem of global constant folding, whose solution requires simultaneous use of chains in both directions. This is because each constant instruction discovered may lead to more folding at the use points of its output variables, and testing an instruction for constant inputs implies an examination of the defining points of those inputs. Put another way, each time an instruction is replaced by a constant, the folding algorithm must recheck all uses of its output variable to see if the using instruction might also be eliminated. Such a check necessarily involves looking at other definitions which can reach the use. The situation is depicted in Figure 17.

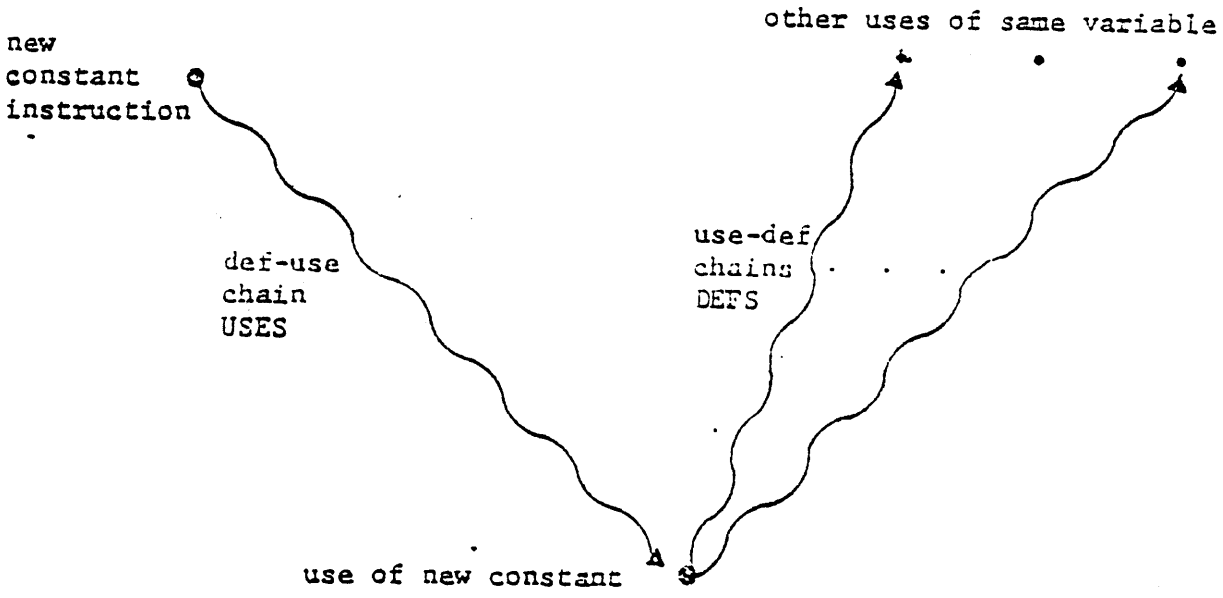


Figure 17. The need for two types of chains in constant folding.

The method implied by the above observation is realized in Algorithm CP. Like Algorithm MK, it uses a workpile to control iterations. A number of set theoretic notations are used in

the informal specification; these have the obvious meanings. The algorithm also uses a subroutine COMPUTE to evaluate constant instructions.

Algorithm CP: Constant Propagation

Input:

1. A program PROG containing instructions of the usual type.
2. A flag CONST(A,i) for each instruction i and input or output ariable A of i. Initially, CONST(A,i) is **true** only if A represents a constant denotation.
3. The chains USES and DEFS.

Ousput:

1. The modified CONST flags.
2. The mapping VAL(A,i) which provides the run-time constant value of variable A at instruction i; VAL(A,i) is defined only if CONST(A,i) is **true**.

Method:

```
begin ¢ start with the trivially constant instructions ¢
   PILE := {x∈PROG | (∀A∈inputs(x) | CONST(A,x))};
   while PILE ≠ φ do
      x := an arbitrary element of PILE;
      PILE := PILE − {x};
      B := output(x);
      for each i ∈ USES(B,x) do
         ¢ check for constant inputs ¢
         conB := true;
         for each y ∈ DEFS(B,i)−{x} while conB do
            if CONST(B,y) andf VAL(B,y)=VAL(B,x)
               then conB := true
               else conB := false
               fi
            od;
         ¢ test the exit condition ¢
         if conB then
            CONST(B,i) := true;
            VAL(B,i) := VAL(B,x);
            ¢ is the instruction now constant? ¢
            if (∀A∈inputs(i) | CONST(A,i)) then
               C := output(i);
               CONST(C,i) := true;
               VAL(C,i) := COMPUTE(i);
               PILE := PILE ∪ {i}
               fi
            fi
         od
      od
   end
```

Although termination and correctness of Algorithm CP are subtle, the interested reader will not find it difficult to establish them. The algorithm is interesting because it serves as a model for many other optimization algorithms. One such will be seen in Section 6.

5. SYMBOLIC INTERPRETATION

The analysis methods presented so far can only solve restricted classes of data flow problems. The algorithms of Section 3 work only for problems which ask whether or not a single event

may (or must) have happened before control reaches some point (in the forward case) or may happen later (in the backward ase). They are not effective for questions about *sequences* of events along control flow paths. Use-definition chain methods are more general, but they too can be imprecise because information is gathered by jumping between uses and definitions rather than by following individual execution paths [Kap78].

The most precise method for gathering global data flow information is *symbolic interpretation* [Weg75, Kin76]. As implied by the name, symbolic interpretation entails executing the program with symbolic values for all variables whose values are indeterminate at compile time. For example, if the value of N in a given FORTRAN program is always 5 but the value of M is read in as data, M would be assigned a symbolic value α. Then after executing the statement

$$L = N*M$$

L will have the (partially) symbolic value 5α.

It should be easy to see that the value numbering method of Section 2 is just symbolic interpretation restricted to straight-line code. As in value numbering, the compiler can uncover useful facts about the relationships among values of program variables at point p by executing the program symbolically up to that point. But there is, of course, a hitch. At conditional transfers of control, the truth value of the condition may depend on symbolic values; that is, it may not be possible to determine at compile time which way control will go at run time. In such cases, interpretation must proceed down *both* paths. But this leads to problems at points where control paths join. If X has value α on one path and β on another, its value after they join must be expressed as "either α or β." In loops, value disjunctions of arbitrary length can be built, as the example in Figure 18 shows.

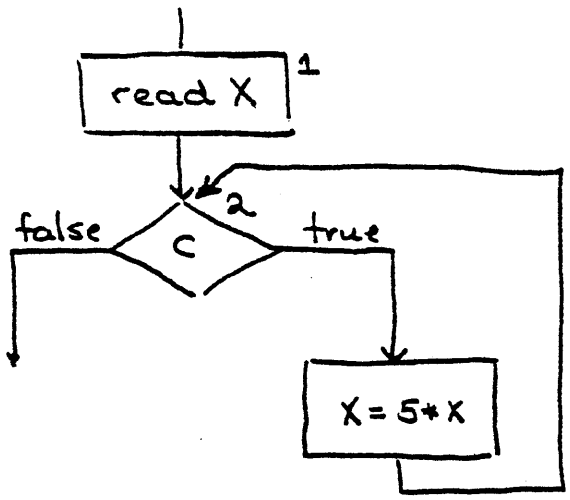


Figure 18. A loop for symbolic interpretation.

Suppose we assign X the value α at block 1; then interpreting around the loop shows that ist value at block 2 can be either α or 5α. Another interpretation adds 25α to the list of alternatives. Clearly, there are infinitely many possible values. Since symbolic interpretation attempts to prove everything it can about a program, it terminates only when it has enumerated all possible values of the properties it is keeping track of, so interpretation would not terminate on this example.

The problem is solved by restricting the application of symbolic interpretation to determining properties from a *well-founded property set* [Weg75]. Simply put, if we take two properties from a well-founded set, their disjunction ("either property α or property β") can be approximated by another property in the set, say γ; furthermore, after finitely many such approximations a limiting property will be reached. For example, suppose we are optimizing a language in which variables may dynamically take on values of three different types: *real*, *integer*, and *character*. Suppose also that the special atomic type *undefined* is used for uninitialized variables and for values resulting from errors. By adding three more types — *number*, *atom*, and *inconsistent* — we can characterize our knowledge of variable types with the well-founded property set shown in Figure 19.

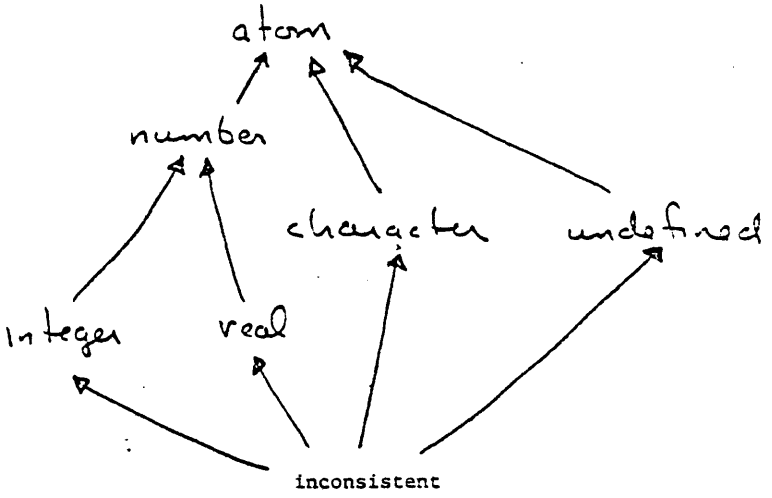


Figure 19. A well-founded property set for variable types.

In this diagram, arcs lead from more specific to less specific information. To determine the result of a disjunction of two distinct types, locate the types in the diagram and find the first type which can be reached from both by following arrows. Thus the disjunction "*real* or *integer*" yields *number*, while "*real* or *undefined*" yields *atom*.

Since the disjunction of a type with itself produces the same type, a stable upper bound must be reached in this set after at most three distinct disjunctions. Thus a symbolic interpreter which terminates only when a steady state is reached will always terminate using this set. In general, symbolic interpretation is guaranteed to terminate when determining properties from a well-founded set on a finite program [Weg75].

To convey the flavor of this method, I will include an adaptation of Wegbreit's simplest interpretation scheme. (More complicated versions, which unroll loops, will not be described.) First we assume a very simple model in which there are only two types of statements, *simple* and *conditional*. A simple statement x has a single successor given by *next*(x), while a

conditional y has two successors: $next_T(y)$, taken when the condition is true, and $next_F(y)$, taken when it is false.

Assume we are dealing with a well-founded property set **P** which has a property disjunction or *join* operation $\vee$ such that, for $p_1, p_2 \in \mathbf{P}$, $p_1 \vee p_2$ is the approximation of "either p_1 or p_2." Furthermore, assume there is a *least general property*, denoted by **0**, such that for any property $p \in \mathbf{P}$, $p \vee \mathbf{0} = p$. In Figure 18, "type=*inconsistent*" is **0**.

Finally, the execution of an elementary statement may change the property which holds after that statement. Let $outprop(x,p)$ be the property which holds after simple statement x is executed, given that property p holds initially. Similar functions $outprop_T(x,p)$ and $outprop_F(x,p)$ give the resultant properties on the true and false branches, respectively, of a conditional.

Algorithm SI: Symbolic Interpretation

Input:

1. A program PROG consisting of instructions with successor fields *next* or $next_T$ and $next_F$.
2. A well-founded property set **P** with join operation $\vee$ and minimal element **0**.
3. The semantic mappings *outprop*, $outprop_T$, and $outprop_F$.

Output: For each statement $x \in \mathbf{P}$, PROP[x], the most specific property provably true on entry to x (within the given framework).

Method:

```
begin
   for each x∈PROG do
      PROP[x] := 0
      od;
   let x0 := the program entry statement;
   PILE := {<x0, 0>};

   while PILE≠φ do
      let z be an arbitrary element in PILE;
      PILE := PILE − {z};
      <x, p> := z;
      oldp := PROP[x];
      PROP[x] := PROP[x] ∨ p;

      while x≠ exit statement and oldp≠PROP[x] do
         if x is a simple statement then
            p := outprop(x,PROP[x]);
            x := next[x];
         else ¢ a conditional; save the false branch ¢
            yF := nextF[x];
            PILE := PILE ∪ {<yF, outpropF(x,PROP[x])>};
            ¢ follow the true branch ¢
            p := outpropT(x,PROP[x]);
            x := nextT[x]
            fi;
         oldp := PROP[x];
         PROP[x] := PROP[x] ∨ p
         od
      od
end
```

Using the well-foundedness of P, it is not too difficult to show that this algorithm terminates. Some unnecessary iterations can be avoided by using a more sophisticated structure for PILE, so that the two pairs $<x, p_1>$ and $<x, p_2>$ are automatically combined into $<x, p_1 \vee p_2>$ when the second is added to a PILE already occupied by the first. The more complicated versions of Algorithm SI that unroll loops for more precision are straightforward extensions [Weg75, Kin76].

If symbolic interpretation is so good, why isn't it used exclusively? The main reason is efficiency. Most problems involve property sets much richer than the one in Figure 18. For example, instead of specifying the type of a single variable, a property might specify the types of *all* program variables. Such property sets give rise to numerous iterations before a steady state is reached. Thus symbolic interpretation is rarely used in compilers. However its suitability for complex problems makes it an important tool for optimization research and program verification [Kin76, CoC77, SuI77, CoH78].

6. OPTIMIZATION OF VERY-HIGH-LEVEL LANGUAGES

I shall conclude this survey with a discussion of some current work on optimization for very-high-level languages, focusing on the SETL project at New York University. SETL is a language based on the theory of sets [Scz73, KeS75]. It has a standard set of fundamental data types (real, integer, character, bit, and strings of characters or bits) along with two structured types -- sets and tuples. It derives its power from its fundamental view of data as sets and mappings (sets of ordered pairs). An introductory treatment of the language may be found in [KeS75].

The SETL implementation identifies two classes of objects, *long* and *short* Both items use a *root word* for their representation. As shown in Figure 20, the first few bits of the root word identify the object type and the rest are used for actual data, in the case of a short object, or control information and a pointer in the case of a long object. A long object's data is contained in an extended *representing block* stored elsewhere and pointed to by the root word.

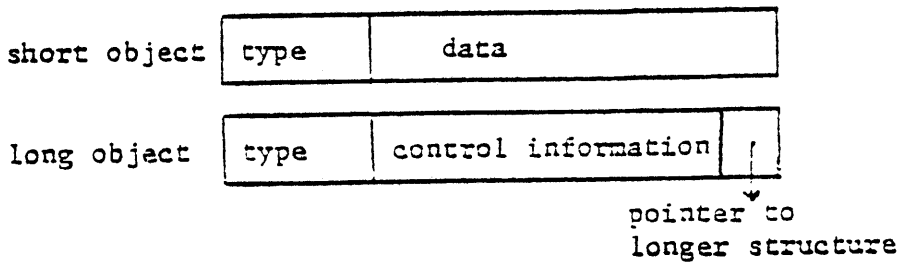


Figure 20. Object representation in SETL.

Currently, SETL uses representing blocks organized as arrays for tuples and hash tables for sets. Individual entries in these blocks are root words for the individual members.

The general unoptimized implementation scheme is as follows. Code is translated into a series of calls to SETL runtime library routines. Each routine implements one SETL primitive in its most general form. In particular, since SETL does not have type declarations, type tests must be made at run time. Consider the primitive

$$s_1 \; eq \; s_2$$

which tests for equality between objects of any type. Even after it is discovered that s_1 and s_2 are both sets, the test is a complex one involving another primitive, the membership test ϵ

$$s_1 \; eq \; s_2 \equiv (\forall x \epsilon s_1 \mid x \epsilon s_2) \& (\forall y \epsilon s_2 \mid y \epsilon s_1)$$

The strategy of the SETL optimizer is to use special knowledge of the program, gleaned through global analysis, to replace as many expensive library calls as possible by in-line *code stubs*, which assume the most common case and test for exceptions, calling the library only when necessary. As an example, consider the expression $x+y$. In the general case, x and y could be sets, integers, tuples, reals, strings, etc. But suppose a global analysis of types determines that x and y are both integers; then the situation is greatly simplified, although we still don't know whether they are long or short integers (long integers require multi-word storage). The code stub assumes, as the most likely case, that both are short integers. It then has the following flavor.

stub: add x and y as short integers;
execute a fast test for overflow or type error;
if test positive **then** call library routine
else record results **fi**

Thus with the aid of global type analysis, the optimizer is able to effect a substantial efficiency gain.

This example leads us naturally to consider the nature of global type analysis. Type analysis was the subject of Tenenbaum's Ph.D thesis [Ten74] and has been subsequently studied by Jones and Muchnick [JoM76] and Kaplan and Ullman [KaU77]. The first step in type analysis is to define an *algebra of type symbols* which is built up from:

a) A number of *atomic type symbols:*
I (integer), R (real), UD (undefined), NS (set of arbitrary elements),
G (general), Z (error), etc.
b) alternation of types:
$t = t_1 \mid t_2 \mid \ldots \mid t_k$
c) set formation:
$t = \{t_1\}$
d) Tuple formation (fixed length):
$t = \langle t_1, t_2, \ldots, t_k \rangle$
e) tuple formation (indefinite length):
$t = [t_1]$

Next we define the rules for determining the output type of an operation given the input types. This is encoded in a transition function F which, for each operation op and input types $t_1, t_2, \ldots, t_n$ of the operands, produces

$$t_o = F_{op}\,(t_1, t_2, \ldots, t_n)$$

where t_o is the output type (or at least the best approximation to it within the algebra). Finally an operation $\vee$, which allows alternation of types at merging paths, is defined; i.e.,

$$t = \bigvee_{i=1}^{k} t_i$$

is the type of an object which has types $t_1, \ldots, t_k$ on k merging paths.

With these definitions, global type determination can be carried out by a direct analog of the use-definition chain algorithm for constant propagation. Although this is the same problem we solved by symbolic interpretation in the last section, use-definition chains permit a more efficient implementation. The workpile is initialized to a set of instructions with clearly defined (or constant) types. Thereafter an instruction is examined whenever a refinement of one of its input types is detected.

Algorithm TA: Type Analysis

Input:

1. A program PROG.
2. A mapping TYPE, such that TYPE(A,x) is the best initial estimate of the type of variable A at x (for most variables this is 'UD').
3. The sets DEFS and USES.

Output: For each instruction x and input or output variable A, TYPE(A,x), a conservative approximation to the most specific type information provably true at x.

Method:

```
begin
   PILE := {x ∈ PROG | (∀A ∈ inputs(x) | TYPE(a,x)≠'UD')};
   while PILE≠φ do
      x := an arbitrary element in PILE;
      PILE := PILE − {x};
      B := output(x);
      for each i ∈ USES(b,x) do
         ¢ recompute type ¢
         oldtype := TYPE(b,i);
         TYPE(B,i) :=   ∨   TYPE(B,y);
                      y ∈ DEFS(B,i)
         if TYPE(B,i)≠oldtype then
            ¢ a type refinement ¢
            TYPE(output(i),i) := F_op(i) applied to the input types of i;
            PILE := PILE ∪ {i}
            fi
         od
      od
   end
```

In his dissertation, Tenenbaum showed how the above type analysis could be enhanced by a backward pass which elicits type information from uses and propagates it back to definition points [Ten74]. Kaplan and Ullman extended this idea to incorporate multiple passes in both directions [KaU77]. It is clear that symbolic interpretation could also be used for type analysis to produce more specific results. I will not have space to treat the numerous other SETL optimizations here. I refer the interested reader to a series of papers [Scz74, Scz75a, Scz75b, Scz75c, Dew77] which lay out most of the methods used by that project; several of these involve automatic or semiautomatic data structure choice. A number of papers treat further SETL optimizations [FoU76, PaS77, Fon77]. In general, the optimization of very-high-level languages should prove a fruitful area for new research and for further application of established techniques.

Acknowledgement

I am grateful to Barry Rosen for several suggestions which substantially improved the paper.

REFERENCES

AhU73 Aho, A. V. and Ullman, J. D., *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling,* Prentice-Hall, Englewood Cliffs, NJ, 1973.

AhU75 Aho, A. V. and Ullman, J. D., "Node listings for reducible flow graphs," *Proc Seventh Annual ACM Symposium on Theory of Computing,* Albuquerque, NM, May 1975, 177-185.

ASU72 Aho, A. V., Sethi, R. and Ullman, J. D., "Code optimization and finite Church-Rosser systems," *Design and Optimization of Compilers,* R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972.

All69 Allen, F. E., "Program optimization," *Annual Review of Automatic Programming* 5, Pergamon, Elmsford, NY, 1969, 239-307.

All70 Allen, F. E., "Control flow analysis," *SIGPLAN Notices,* 5,7, July 1970, 1-19.

All71 Allen, F. E., "A basis for program optimization," *Proc. IFIP Congress 71,* North-Holland Publishing Co., Amsterdam, 1971, 385-391.

All74 Allen, F. E., "Interprocedural data flow analysis," *Proc IFIP Congress 74,* North-Holland Publishing Co., Amsterdam, 1974, 398-402.

AlC72a Allen, F. E. and Cocke, J., "A catalogue of optimizing transformations," *Design and Optimization of Compilers,* R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, 1-30.

AlC72b Allen, F. E. and Cocke, J., "Graph theoretic constructs for program control flow analysis," IBM Research Report RC3923, Thomas J. Watson Research Center, Yorktown Heights, NY, July 1972.

AlC76 Allen, F. E. and Cocke, J., "A program data flow analysis procedure," *Comm. Acm,* 19, 3, March 1976, 137-147.

AsM71 Ashcroft, E., and Manna, Z., "The translation of 'go to' programs into 'while' programs," *Proc. IFIP Congress 71,* North-Holland Publishing Co., Amsterdam, 1971.

BaJ78a Babich, W.A., and Jazayeri, M. "The method of attributes for data flow analysis (Part I. Exhaustive analysis)," *Acta Informatica* 10, 1978, 245-264.

BaJ78b Babich, W.A., and Jazayeri, M. "The method of attributes for data flow analysis (Part II. Demand analysis)," *Acta Informatica,* 10, 1978, 265-272.

Bar78 Barth, J. M., "A practical interprocedural data flow analysis program," *Comm. ACM,* 21, 9, September 1978, 724-736.

Ban79 Banning, J. "An efficient way to find the side effects of procedure calls and aliases of variables," *Conf. Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,*, San Antonio, Texas, January 1979, 29-41.

Bea72 Beatty, J. C. "An axiomatic approach to code optimization for expressions," *J. ACM,* 19, 4, October 1972, 613-640.

Bea74 Beatty, J., "A register assignment algorithm for generation of highly optimized object code," *IBM J. Res. Develop.,* 18, 1, January 1974, 20-39.

BoJ66 Böhm,C. and Jacopini, G., "Flow diagrams, Turning machines, and languages with only two formation rules," *Comm. ACM,* 19, 5, May 1966, 366-371.

Coc70 Cocke, J., "Global common subexpression elimination," *SIGPLAN Notices,* 5, 7, July 1970, 20-24.

CoM69 Cocke, J. and Miller, R. E., "Some analysis techniques for optimizing computer programs," *Proc 2nd International Conference on System Sciences,* Hawaii, 1969, 143-146.

CoK76 Cocke, J. and Kennedy, K. "Profitability computations on program flow graphs," *Computers and Math. with Applications,* 2, 1976, Pergamon, 145-159.

CoK77 Cocke, J. and Kennedy, K., "An algorithm for reduction of operator strength," *Comm. ACM* 20,11, November 1977, 850-856.

CoS70 Cocke, J. and Schwartz, J. T., *Programming Languages and Their Compilers: Preliminary Notes,* Computer Science Dept., New York University, 1970.

CoC77 Cousot, P. and Cousot, R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Conf. Record of the Fourth ACM Symposium on the Principles of Programming Languages,* Los Angeles, January 1977, 238-252.

CoH78 Cousot, P. and Halbwachs, N., "Automatic discovery of linear restraints among variables of a program," *Conf. Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,* Tucson, Arizona, January 1978, 84-96.

Dew77 Dewar, R. B. K., Grand, A., Liu, S.-C., Schonberg, E., and Schwartz, J. T., "Programming by refinement as exemplified by the SETL representation sublanguage," draft, Dept. of Computer Science, New York University, 1977.

Ear74 Earnest, C. "Some topics in code optimization," *J. ACM,* 21,1, January 1974, 75-102.

EBA72 Earnest, C., Balke, K. and Anderson, J., "Analysis of graphs by ordering of nodes," *J. ACM,* 19, 1, January 1972, 23-42.

FKZ75 Farrow, R., Kennedy, K. and Zucconi, L., "Graph grammars and global program data flow analysis," *Proc. 17th Annual IEEE Symposium on Foundations of Computer Science,* Houston, November 1975.

Fon77 Fong, A. C., "Generalized common subexpressions in very high level languages," *Conf. Record of the Fourth ACM Symposium on Principles of Programming Languages,* Los Angeles, January 1977.

FoU76 Fong, A. C., and Ullman, J. D., "Induction variables in very high level languages," *Conf. Record of the Fourth ACM Symposium on Principles of Programming Languages,* Atlanta, January 1976.

GrW76 Graham, S. L. and Wegman, M., "A fast and usually linear algorithm for global flow analysis," *J. ACM,* 23, 1, January 1976, 172-202.

Har75 Harrison W., "Compiler analysis of the value ranges of variables," IBM Research Report RC5544, Thomas J. Watson Research Center, Yorktown Heights, NY, July 1975.

HeU72 Hecht, M. S. and Ullman, J. D., "Flow graph reducibility," *SIAM J. Computing* 1,2, June 1972, 188-202.

HeU74 Hecht, M. S. and Ullman, J. D., "Characterizations of reducible flow graphs," *J. ACM* , 21, 3, July 1974, 367-375.

HeU75 Hecht, M. S. and Ullman, J. D., "A simple algorithm for global data flow analysis," *SIAM J. Computing,* 4, 4, December 1975, 519-532.

Jaz75 Jazayeri, M., "Live variable analysis, attribute grammars, and program optimization," draft, Dept. of Computer Science, University of North Carolina, Chapel Hill, N.C., March 1975.

JoM76 Jones, N. and Muchnick, S., "Binding time optimization in programming languages," *Conf. Record of the Third ACM Symposium on Principles of Programming Languages,*" Atlanta, January 1976, 77-94.

KaU76 Kam, J. B. and Ullman, J. D., "Global data flow analysis and iterative algorithms," *J. ACM,* 23, 1, January 1976, 158-171.

Kap78 Kaplan, M. A., "Relational data flow analysis," TR-243, Dept. of E.E. and Computer Science, Princeton University, April 1978.

KaU77 Kaplan, M. A. and Ullman, J. D., "A general scheme for the automatic inference of variable types," TR-226, Dept. of Electrical Engineering, Princeton University, June 1977.

Ken71 Kennedy, K., "A global flow analysis algorithm," *Intern. J. Computer Math,* Section A, 3, December 1971, 5-15.

Ken75a Kennedy, K., "Node listings applied to data flow analysis," *Conf. Record of the Second ACM Symposium on Principles of Programming Languages,* Palo Alto, CA, January 1975, 10-21.

Ken75b Kennedy, K. "Use-definition chains with applications," Technical Report 476-093-9 Dept. of Mathematical Sciences, Rice University, Houston, April 1975.

Ken76 Kennedy, K., "A comparison of two algorithms for global data flow analysis," *SIAM J. Computing,* 5, 1, March 1976, 158-180.

KeS75 Kennedy, K. and Schwartz, J. T., "An introduction to the set theoretic language SETL," *Computers and Math. with Applications,* 1, Pergamon Press, 1975, 97-119.

KeZ77 Kennedy, K. and Zucconi, L., "Application of a graph grammar for program control flow analysis," *Conf. Record of the Fourth ACM Symposium on the Principles of Programming Languages,* Los Angeles, January 1977, 72-85.

KeZ78 Kennedy, K. and Zucconi, L. "Basic block optimization in Model," draft report, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, 1978.

Kil73 Kildall, G. A., "A unified approach to global program optimization," *Conf. Record ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages,* Boston, October 1973, 194-206.

Kin76 King, J. C., "Symbolic execution and program testing," *Comm. ACM,* 19, 7, July 1976, 385-394.

Knu68 Knuth, D. E., "Semantics of contest-free languages," *Math. Systems Theory,* 2, 1968, 127-145.

Knu71 Knuth, D. E., "An empirical study of FORTRAN programs," *Software-Practice and Experience,* 1,2, 1971, 105-134.

LoM69 Lowry, E. S. and Medlock, C. W., "Object code optimization," *Comm. ACM,* 12, 1, January 1969, 13-22.

MaT75 Markowsky, G. and Tarjan, R. E., "Lower bounds on the lengths of node sequences in directed graphs," IBM Research Report RC5477, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, July 1975.

NeA75 Neel, D. and Amirchahy, M., "Removal of invariant statements from nested loops in a single effective compiler pass," *SIGPLAN Notices,* 10, 3, March 1975, 87-96.

PaS77 Paige, B. and Schwartz, J. T., "Expression continuity and the formal differentiation of algorithms," *Conf. Record of the Fourth ACM Symposium on principles of Programming Languages,* Los Angeles, January 1977, 58-71.

Ros77 Rosen, B. K., "High-level data flow analysis," *Comm. ACM,* 20, 10, October 1977, 712-724.

Ros78 Rosen, B. K., "Monoids for rapid data flow analysis," *Conf. Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,* Tucson, Arizona, January 1978, 47-59.

Ros79 Rosen, B. K., "Data flow analysis for procedural languages," *J. ACM,* 26, 2, April 1979, 322-344.

Sch73 Schaefer, M., *A Mathematical Theory of Global Program Optimization,* Prentice-Hall, Englewood Cliffs, NJ, 1973.

Sck75 Schneck, P. B., "Movement of implicit parallel and vector expressions out of program loops," *SIGPLAN Notices,* 10, 3, March 1975, 103-106.

Scz73 Schwartz, J. T., "On programming, an interim report on the SETL project," Computer Science Dept., New York University, October 1973.

Scz74 Schwartz, J. T., "Automatic and semiautomatic optimization of SETL," *SIGPLAN Notices*, 9, 4, April 1974, 43-49.

Scz75a Schwartz, J. T., "Optimization of very high level languages I; value transmission and its corollaries," *J. Computer Languages* 1, Pergamon Press, 1975, 161-194.

Scz75b Schwartz, J. T., "Optimization of very high level languages II, deducing relationships of inclusion and membership," *J. Computer Languages*, 1, Pergamon Press, 1975, 197-218.

Scz75c Schwartz, J. T., "Automatic data structure choice in a language of very high level," *Comm. ACM*, 18,12, December 1975, 722-728.

Set74 Sethi, R., "Testing for the Church-Rosser property," *J. ACM* 21, 4, October 1974, 671-679.

SeU70 Sethi, R. and Ullman, J. D., "The generation of optimal code for arithmetic expressions," *J. ACM*, 17, 4, October 1970, 715-728.

Sta76 Standish, T. A. et. al., "The Irvine program transformation catalogue," Dept. of Information and Computer Science, Univ. California at Irvine, January 1976.

SuI77 Suzuki, N. and Ishihata, K., "Implementation of an array bound checker," *Conf. Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, January 1977.

Tar75a Tarjan, R. E., "Applications of path compression on balanced trees," Technical Report STAN-75-512, Computer Science Dept. Stanford University, Stanford, CA, 1975.

Tar75b Tarjan, R. E., "Solving path problems on directed graphs," Technical Report STAN-75-528, Computer Science Dept., Stanford University, Stanford, CA 1975.

Ten74 Tenenbaum, A., "Automatic type analysis in a very high level language," PhD. Thesis, Computer Science Dept., New York Univeristy, October 1974.

Weg75 Wegbreit, B., "Property extraction in well-founded property sets," *IEEE Trans. on Software Engineering*, SE-1, 3, September 1975, 270-285.

Wul75 Wulf, W., Johnsson, R. K., Weinstock, C. B., Hobbs, S. O., and Geschke, C. M., *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.

Ull73 Ullman, J. D., "Fast algorithms for the elimination of common subexpressions," *Acta Informatica*, 2, 1973, 191-213.