A UNIFIED APPROACH TO GLOBAL PROGRAM OPTIMIZATION

Gary A. Kildall

Computer Science Group
Naval Postgraduate School
Monterey, California

Abstract

A technique is presented for global analysis of program structure in order to perform compile time optimization of object code generated for expressions. The global expression optimization presented includes constant propagation, common subexpression elimination, elimination of redundant register load operations, and live expression analysis. A general purpose program flow analysis algorithm is developed which depends upon the existence of an "optimizing function." The algorithm is defined formally using a directed graph model of program flow structure, and is shown to be correct. Several optimizing functions are defined which, when used in conjunction with the flow analysis algorithm, provide the various forms of code optimization. The flow analysis algorithm is sufficiently general that additional functions can easily be defined for other forms of global code optimization.

1. INTRODUCTION

A number of techniques have evolved for the compile-time analysis of program structure in order to locate redundant computations, perform constant computations, and reduce the number of store-load sequences between memory and high-speed registers. Some of these techniques provide analysis of only straight-line sequences of instructions [5,6,9,14, 17,18,19,20,27,29,34,36,38,39,43,45,46], while others take the program branching structure into account [2,3,4,10,11,12,13,15,23,30,32,33,35]. The purpose here is to describe a single program flow analysis algorithm which extends all of these straight-line optimizing techniques to include branching structure. The algorithm is presented formally and is shown to be correct. Implementation of the flow analysis algorithm in a practical compiler is also discussed.

The methods used here are motivated in the section which follows.

2. CONSTANT PROPAGATION

A fairly simple case of program analysis and optimization occurs when constant computations are evaluated at compile-time. This process is referred to as "constant propagation," or "folding." Consider the following skeletal ALGOL 60 program:

```
begin integer i,a,b,c,d,e;
a:=1; c:=0; . . .
for i:=1 step 1 until 10 do
    begin b:=2; . . .
    d:=a+b; . . .
    e:=b+c; . . .
    c:=4;   . . .
    end
end
```

This program is represented by the directed graph shown in Figure 1 (ignoring calculations which control the for-loop). The nodes of the directed graph represent sequences of instructions containing no alternate program branches, while the edges of the graph represent program control flow possibilities between the nodes at execution-time.

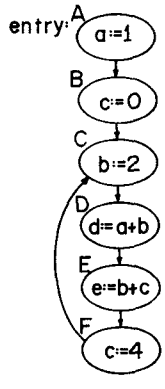

Figure 1. A program graph corresponding to an ALGOL 60 program containing one loop.

For purposes of constant propagation, it is convenient to associate a "pool" of propagated constants with each node in the graph. The pool is a set of ordered pairs which indicate variables which have constant values when the node is encountered. Thus, the pool of constants at node B, denoted by P_B, consists of the single element (a,1) since the assignment a:=1 at node A must occur before node B is encountered during execution of the program.

The fundamental global analysis problem is that of determining the pool of propagated constants for each node in an arbitrary program graph. By inspection of the graph of Figure 1, the pool of constants at each node is

$$P_A = \emptyset \qquad P_D = \{(a,1),(b,2)\}$$
$$P_B = \{(a,1)\} \qquad P_E = \{(a,1),(b,2),(d,3)\}$$
$$P_C = \{(a,1)\} \qquad P_F = \{(a,1),(b,2),(d,3)\}\ .$$

In the general case, P_N could be determined for each node N in the graph as follows. Consider each path $(A,p_1,p_2,\ldots,p_n,N)$ from the entry node A to the node N. Apply constant propagation throughout this path to obtain a set of propagated constants at node N for this path only. The intersection of the propagated constants determined for each path to N is then the set of constants which can be assumed for optimization purposes, since it is not known which of the paths will be taken at execution-time.

The pool of propagated constants at node D of Figure 1, for example, can be determined as follows. A path from the entry node A to the node D is (A,B,C,D). Considering only this path, the "first approximation" to P_D is

$$P_D^1 = \{(a,1),(b,2),(c,0)\}$$

A longer path from A to D is (A,B,C,D,E,F,C,D) which results in the pool

$$P_D^2 = \{(a,1),(b,2),(c,4),(d,3),(e,2)\}$$

corresponding to this particular path to D. Successively longer paths from A to D can be evaluated, resulting in P_D^3, P_D^4, ..., P_D^n for arbitrarily large n. The pool of propagated constants which can be assumed no matter which flow of control occurs is the set of constants common to all P_D^i; that is,

$$P_D = \bigcap_{\forall\, i} P_D^i$$

This procedure, however, is not effective since the number of such paths may have no finite bound in the case of an arbitrary directed graph. Hence, the procedure would not necessarily halt. The purpose of the algorithm of the following section is to compute this intersection in a finite number of steps.

3. A GLOBAL ANALYSIS ALGORITHM

The analysis of the program graph of Figure 1 suggests a solution to the global constant propagation problem. Considering node C, the first approximation to P_C is given by propagating constants along the path (A,B,C), resulting in

$$P_C^1 = \{(a,1),(c,0)\}.$$

Based upon this approximate pool, the first approximations to subsequent nodes can be determined:

$$P_D^1 = \{(a,1),(c,0),(b,2)\},$$
$$P_E^1 = \{(a,1),(c,0),(b,2),(d,3)\}$$
$$P_F^1 = \{(a,1),(c,0),(b,2),(d,3),(e,2)\}.$$

Using P_F^1, the constant pool resulting from node F entering node C is

$$P = \{(a,1),(b,2),(d,3),(e,2),(c,4)\}.$$

Note, however, that since

$$P_C = \bigcap_{\forall\, i} P_C^i$$

it follows that $P_C \subseteq P_C^1 \cap P_C^2$. Thus, rather than assuming $P_C^2 = P$, the second approximation to P_C is taken as

$$P_C^2 = P_C^1 \cap P = P_C^1 \cap \{(a,1),(b,2),(d,3),(e,2),(c,4)\} = \{(a,1)\}.$$

Using P_C^2, the circuit through the loop past C is traced once again. The next approximation at subsequent nodes can then be determined based upon P_C^2:

$$P_D^2 = P_D^1 \cap \{(a,1),(b,2)\} = \{(a,1),(b,2)\},$$
$$P_E^2 = P_E^1 \cap \{(a,1),(b,2),(d,3)\} = \{(a,1),(b,2),(d,3)\}\ ,$$
$$P_F^2 = P_F^1 \cap \{(a,1),(b,2),(d,3)\} = \{(a,1),(b,2),(d,3)\}.$$

Continuing around the loop once again from node F to node C, the third approximate pool P_C^3 is determined as

$$P_C^3 = P_C^2 \cap \{(a,1),(b,2),(d,3)\} = \{(a,1)\}.$$

Clearly, no changes to subsequent approximate pools will occur if the circuit is traversed again since $P_C^3 = P_C^2$, and the effect of P_C^2 on the pools in the circuit has already been investigated. Thus, the analysis stops, and the last approximate pools at each node are taken as the final constant pools. Note that these last approximations correspond to the constant pools determined earlier by inspection.

Based upon these observations, it is possible to informally state a global analysis algorithm.

a. Start with an entry node in the program graph, along with a given entry pool corresponding to this entry node. Normally, there is only one entry node, and the entry pool is empty.

b. Process the entry node, and produce optimizing information (in this case, a set of propagated constants) which is sent to all immediate successors of the entry node.

c. Intersect the incoming optimizing pools with that already established at the successor nodes (if this is the first time the node is encountered, assume the incoming pool as the first approximation and continue processing).

d. Considering each successor node, if the amount of optimizing information is reduced by this intersection (or if the node has been encountered for the first time) then process the successor in the same manner as the initial entry node (the order in which the successor

nodes are processed is unimportant).

In order to generalize the above notions, it is useful to define an "optimizing function" f which maps an "input" pool, along with a particular node, to a new "output" pool. Given a particular set of propagated constants, for example, it is possible to examine the operation at a particular node and determine the set of propagated constants which can be assumed after the node is executed. In the case of constant propagation, the function can be informally stated as follows. Let V be a set of variables, let C be a set of constants, and let $\underline{N}$ be the set of nodes in the graph being analyzed. The set $U = V \times C$ represents ordered pairs which may appear in any constant pool. In fact, all constant pools are elements of the power set of U (i.e., the set of all subsets of U), denoted by $P(U)$. Thus,

$$f: \underline{N} \times P(U) \rightarrow P(U),$$

where $(v,c) \in f(N,P) \iff$

a. $(v,c) \in P$ and the operation at node N does not assign a new value to the variable v, or

b. the operation at node N assigns an expression to the variable v, and the expression evaluates to the constant c, based upon the constants in P.

Consider the graph of Figure 1, for example. The optimizing function can be applied to node A with an empty constant pool resulting in

$$f(A,\emptyset) = \{(a,1)\}.$$

Similarly, the function f can be applied to node B with $\{(a,1)\}$ as a constant pool yielding

$$f(B, \{(a,1)\}) = \{(a,1),(c,0)\}.$$

Note that given a particular path from the entry node A to an arbitrary node $N \in \underline{N}$, the optimizing pool which can be assumed for this path is determined by composing the function f up to the last node of the path. Given the path (A,B,C,D), for example,

$$f(C,f(B,f(A,\emptyset))) = \{(a,1),(c,0),(b,2)\}$$

is the constant pool at D for this path.

The pool of optimizing information which can be assumed at an arbitrary node N in the graph being analyzed, independent of the path taken at execution time, can now be stated formally as

$$P_N = \bigcap_{x \in F_N} x$$

where

$$F_N = \{f(p_n,f(p_{n-1},\ldots,f(p_1,P))\ldots) \mid (p_1,p_2,\ldots,p_n,N) \text{ is a path from an entry node } p_1 \text{ with corresponding entry pool P to the node N}\}.$$

Before formally stating the global analysis algorithm, it is necessary to clarify the fundamental notions.

A finite directed graph $G = \langle \underline{N},\underline{E} \rangle$ is an arbitrary finite set of "nodes" $\underline{N}$ and "edges" $\underline{E} \subseteq \underline{N} \times \underline{N}$. A "path" from A to B in G, for $A,B \in \underline{N}$, is a sequence of nodes $(p_1,p_2,\ldots p_k) \ni p_1 = A$ and $p_k = B$, where $(p_i,p_{i+1}) \in \underline{E}$ $\forall i$, $1 \leq i < k$. The "length" of a path $(p_1,p_2,\ldots,p_k)$ is k-1.

A "program graph" is a finite directed graph G along with a non-empty set of "entry nodes" $\mathcal{E} \subseteq \underline{N}$ such that given $N \in \underline{N}$ $\exists$ a path $(p_1,\ldots,p_n) \ni p_1 \in \mathcal{E}$ and $p_n = N$ (i.e., there is a path to every node in the graph from an entry node).

The set of "immediate successors" of a node N is given by

$$I(N) = \{N' \in \underline{N} \mid \exists\ (N,N') \in \underline{E}\}.$$

Similarly, the set of "immediate predecessors" of N is given by

$$I^{-1}(N) = \{N' \in \underline{N} \mid \exists (N',N) \in \underline{E}\}.$$

Let the finite set $\underline{P}$ be the set of all possible optimizing pools for a given application (e.g., $\underline{P} = P(U)$ in the constant propagation case, where $\underline{U} = V \times C$), and $\wedge$ be a "meet" operation with the properties

$$\wedge: \underline{P} \times \underline{P} \rightarrow \underline{P},$$

$$x \wedge y = y \wedge x \text{ (commutative)},$$

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z \text{ (associative)},$$

where $x, y, z \in \underline{P}$. The set $\underline{P}$ and the $\wedge$ operation define a finite meet-semilattice.

The $\wedge$ operation defines a partial ordering on $\underline{P}$ given by

$$x \leq y \iff x \wedge y = x \ \forall x,y \in \underline{P}.$$

Similarly,

$$x < y \iff x \leq y \text{ and } x \neq y.$$

Given $X \subseteq \underline{P}$, the generalized meet operation $\bigwedge_{x \in X} x$ is defined simply as the pairwise application of $\wedge$ to the elements of X. $\underline{P}$ is assumed to contain a "zero element" $\underline{0} \ni \underline{0} \leq x$ $\forall x \in P$. An augmented set $\underline{P}'$ is constructed from $\underline{P}$ by adding a "unit element" $\underline{1}$ with the properties $\underline{1} \notin \underline{P}$ and $\underline{1} \wedge x = x$ $\forall x \in \underline{P}$; $\underline{P}' = \underline{P} \cup \{\underline{1}\}$. It follows that $x < \underline{1}$ $\forall x \in \underline{P}$.

An "optimizing function" f is defined

$$f: \underline{N} \times \underline{P} \rightarrow \underline{P}$$

and must have the homomorphism property:

$$f(N,x \wedge y) = f(N,x) \wedge f(N,y),\ N \in \underline{N},\ x,y \in \underline{P}.$$

Note that $f(N,x) < \underline{1}$ $\forall N \in \underline{N}$ and $x \in \underline{P}$.

The global analysis algorithm is now stated:

Algorithm A. Analysis of each particular program graph G depends upon an "entry pool set" $\underline{\mathcal{E}} \subseteq \mathcal{E} \times \underline{P}$, where $(e,x) \in \underline{\mathcal{E}}$ if $e \in \mathcal{E}$ is an entry node with

corresponding entry optimizing pool $x \in \underline{P}$.

A1[initialize]	$L \leftarrow \underline{\mathcal{E}}$
A2[terminate ?]	If $L = \emptyset$ then halt.
A3[select node]	Let $L' \in L$, $L' = (N,P_i)$ for some $N \in \underline{N}$ and $P_i \in \underline{P}$, $L \leftarrow L - \{L'\}$
A4[traverse?]	Let P_N be the current approximate pool of optimizing information associated with the node N (initially, $P_N = \underline{1}$). If $P_N \leq P_i$ then go to step A2.
A5[set pool]	$P_N \leftarrow P_N \wedge P_i$, $L \leftarrow L \cup \{(N',f(N,P_N)) \mid N' \in I(N)\}$.
A6[loop]	Go to step A2.

For purposes of constant propagation, $\underline{P} = P(U)$, where $U = V \times C$, as before. The meet operation is $\cap$, and the less-than-or-equal relation is $\subseteq$. Note that the zero element in $P(U)$ is $\emptyset \in P(U)$. The unit element in $P(U)$ is U itself. The algorithm requires a new unit element, however, which is not in $P(U)$. The new unit element is constructed as follows: let δ be a symbol not in U, and let $\underline{U} = U \cup \{\delta\}$. It follows that $\underline{U} \cap x = x \; \forall x \in P(U)$ and $\underline{U} \notin P(U)$. Thus, $\underline{P}' = \underline{P} \cup \{\underline{U}\}$ is obtained from $\underline{P}$ by adding a unit element $\underline{U}$. As demonstrated in the proof in Theorem 2, the addition of the symbol δ to U causes the algorithm A to consider each node in the program graph at least once.

Appendix A shows the analysis of the program graph of Figure 1 using the entry pool set $\underline{\mathcal{E}} = \{(A,\emptyset)\}$.

Theorem 1. The algorithm A is finite.
Proof. The algorithm A terminates when $L = \emptyset$. Each evaluation of step A3 removes an element from L, and elements are added to L only in step A5. Thus, A is finite if the number of evaluations of step A5 is finite. Informally, each evaluation of step A5 reduces the "size" of the pool P_N at some node N. Since the size cannot be less than $\underline{0}$, the process must be finite. Formally, step A5 is performed only when $P_N \neq P_N \wedge P_i$. But $(P_N \wedge P_i) \wedge P_N = P_N \wedge P_i \Rightarrow P_N \wedge P_i \leq P_N$, and $P_N \wedge P_i \neq P_N \Rightarrow P_N \wedge P_i < P_N$. Thus, the approximate pool P_N at node N can be reduced at most to $\underline{0}$ since $P_N \leftarrow P_N \wedge P_i$. Further, since the first approximation to P_N is $\underline{1}$ and the lattice is finite, it follows that step A5 can be performed only a finite number of times. Thus A is finite ●

An upper bound on the number of steps in the algorithm A can easily be determined. Let n be the cardinality of $\underline{N}$ and $h(\underline{P}')$ be a function of $\underline{P}'$ (which, in turn, may be a function of n) providing the maximum length of any chain between $\underline{1}$ and $\underline{0}$ in $\underline{P}'$. Step A5 can be executed a maximum of $h(\underline{P}')$ times for any given node. Since there are n nodes in the program graph, step A5 can be performed no more than $n \cdot h(\underline{P}')$ times.

In the case of constant propagation, for example, let u be the cardinality of U. The size of U varies directly with the number of nodes n. In addition, the maximum length of any chain $u_1, u_2, \ldots, u_k$ such that $u_1 = U$ and $u_k = \emptyset$, where $u_1 \supset u_2 \supset u_3 \ldots \supset u_k$ is u. Thus, $h(P(U)) = u$; and the theoretical bound is $n \cdot u$. Since u varies directly with n, it follows that the order of the algorithm A is no worse than n^2.

The correctness of the algorithm A is guaranteed by the following theorem.

Theorem 2. Let $F_N = \{f(p_n,f(p_{n-1},\ldots, f(p_1,P))..) \mid (p_1,\ldots,p_n,N)$ is a path from an entry node p_1 with corresponding entry pool P to the node N$\}$. Further, let

$$X_N = \bigwedge_{x \in F_N} x$$

corresponding to a particular program graph G, set $\underline{P}'$, and optimizing function f, which satisfy the conditions of the algorithm A. If P_N is the final approximate pool associated with node N when A halts, then $P_N = X_N \;\; \forall N \in N$.

Theorem 2 thus relates the final output of the algorithm to the intuitive results which were developed earlier. The proof of Theorem 2 is given in Appendix B.

An interesting side-effect of Theorem 2 is that the order of choice of elements from L in step A3 is arbitrary, as given in the following corollary.

Corollary 1. The final pool P_N associated with each node $N \in \underline{N}$ upon termination of the algorithm A is uniquely determined, independent of the order of choice of L' from L in step A3.
Proof. This corollary follows immediately, since the proof of Theorem 2 in Appendix B is independent of the choice of L' ●

Since the choice of L' from L in step A3 is arbitrary, it is interesting to investigate the effects of the selection criteria upon the algorithm. The number of steps to the final solution is clearly affected by this choice. No selection method has been established, however, to maximize this convergence rate. One might also notice that by treating accesses to L as critical sections in steps A3 and A5, the elements of L can be processed in parallel. That is, independent processes can be started in step A3 to analyze all elements of L.

It is important to note at this point that the algorithm A allows one to ignore the global analysis, and concentrate upon development of straight-line code optimizing functions. That is, if an optimizing function f can be constructed for optimizing a sequence of code containing no alternative branches, then the algorithm A can be invoked to perform the branch analysis, as long as f satisfies the conditions of the algorithm.

4. COMMON SUBEXPRESSION ELIMINATION

Global common subexpression elimination involves the analysis of a program's structure in order to detect and eliminate calculations of redundant expressions. A fundamental assumption is that it requires less execution time to store the result of a previous computation and load this value when the redundant expression is encountered.

As an example, consider the simple sequence of expressions:

... r:=a+b; ... r+x ... (a+b)+x ...

which could occur as part of an ALGOL 60 program. Figure 2 shows this sequence written as a directed graph. Note that the redundant expression (a+b) at node V is easily recognized. The entire expression (a+b)+x at node V is redundant, however, since r has the same value as a+b at node U, and r+x is computed at node U ahead of node V. It is only necessary to describe an optimizing function f which detects this situation for straight-line code; the algorithm A will make the function globally applicable.

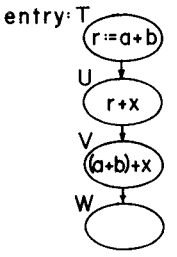

Figure 2. An acyclic program graph representing a simple computation sequence.

A convenient representation for the optimizing pool in the case of common subexpression elimination is a partition of a set of expressions. The expressions in the partition at a particular node are those which occur before the node is encountered at execution-time.

The optimizing function for common subexpression elimination manipulates the equivalence classes of the partition. Two expressions are placed into the same class of the partition if they are known to have equivalent values. Considering Figure 2, for example, the set of expressions which are evaluated before node T is encountered is empty; thus, $P_T = \emptyset$. The expressions evaluated before node U are exactly those which occur at node T, including partial computations. The set of (partial) computations at node T is {a,b,a+b,r}. Since r takes the value of a+b at node T, r is known to be equivalent to a+b. Thus, P_U ={a|b|a+b,r}, where "|" separates the equivalence classes of the pool. Similarly, P_V = {a|b|a+b,r|x|r+x} and P_W = {a|b|a+b,r|x|r+x|(a+b)+x}. The expression a+b at node V is redundant since a+b is in the pool P_V.

Note, however, that the redundant expression (a+b)+x at node V is not readily detected. This is due to the fact that r+x was computed at node U and, as noted above, the evaluation of r+x is the same as evaluation of (a+b)+x at node U. In order to account for this in the output optimizing pool, (a+b)+x is added to the same class as r+x. Thus, P_V becomes

{a|b|a+b,r|x|r+x,(a+b)+x}.

This process is called "structuring" an optimizing pool. Structuring consists of adding any expressions to the partition which have operands equivalent to the one which occurs at the node being considered. The entire expression (a+b)+x at node V is then found to be redundant since the structured pool P_V contains a class with (a+b)+x.

An optimizing function $f_1(N,P)$ for common subexpression elimination can now be informally stated.

1. Consider each partial computation e in the expression at node $N \in \underline{N}$.
2. If the computation e is in a class of P then e is redundant; otherwise
3. create a new class in P containing e and add all (partial) computations which occur in the program graph and which have operands equivalent to e (i.e., structure the pool P).
4. If N contains an assignment d:=e, remove from P all expressions containing d as a subexpression. For each expression e' in P containing e as a subexpression, create e'' with d substituted for e, and place e'' in the class of e'.

The meet operation $\wedge$ of the algorithm A must be defined for common subexpression elimination. Since the optimizing pools in $\underline{P}'$ are partitions of expressions, the natural interpretation is as intersection by classes, denoted by $\overset{*}{\wedge}$. That is, given $P_1, P_2 \in \underline{P}'$, $P = P_1 \overset{*}{\wedge} P_2$ is defined as follows.

Let

$$C = \bigcup_{p \in P_1} p \cap \bigcup_{p \in P_2} p$$

$$\text{and } P(c) = P_1(c) \cap P_2(c) \ \forall c \in C.$$

C is the set of expressions common to both P_1 and P_2, while $P_1(c)$ and $P_2(c)$ are the classes of c in P_1 and P_2, respectively. Thus, the class of each $c \in C$ in the new partition P is derived from P_1 and P_2 by intersecting the classes $P_1(c)$ and $P_2(c)$. For example, if P_1 = {a,b|d,e,f} and P_2 = {a,c|d,f,g} then C = {a,d,f} and $P_1 \overset{*}{\wedge} P_2$ = {a|d,f}.

It is easily shown that $\overset{*}{\wedge}$ has the properties required of the meet operation; hence, a "refinement" relation is defined:

$$P_1 \overset{*}{\leq} P_2 \iff P_1 \overset{*}{\wedge} P_2 = P_1.$$

That is, $P_1 \overset{*}{\leq} P_2$ if and only if P_1 is a refinement of P_2. The refinement relation provides the ordering required on the set $\underline{P}'$ for the algorithm A.

The function f_1 can be stated formally, and shown to have the homomorphism property required by the global analysis algorithm [33]:

$$f_1(N,P_1 \overset{*}{\wedge} P_2) = f_1(N,P_1) \overset{*}{\wedge} f_1(N,P_2).$$

Before considering an example of the use of f_1 with the algorithm A, the function f_1 is extended to combine constant propagation with common subexpression elimination.

5. CONSTANT PROPAGATION AND COMMON SUBEXPRESSION ELIMINATION

The common subexpression elimination optimizing function f_1 of Section 4 can easily be extended to include constant propagation. Consider, for example, the following segment of an ALGOL 60 program:

... u:=20; ... v:=30; ... u+v ... x:=10;

... y:=40; ... x+y ... y-x ...

Figure 3 shows a program graph representing this segment. Assume the entry pool is empty; i.e., $P_B = \emptyset$. The analysis proceeds up to node E as before, resulting in

$$P_E = \{u,20|v,30\}.$$

Note that u and v are both propagated constants in P_E since they are both in classes containing constants. If the expression u+v at node E is processed as in f_1, the output pool is

$$\{u,20|v,30|u+v\}.$$

Noting that u and v are in classes with constants, then u+v must be the propagated constant 20+30 = 50. Hence, the constant 50 is placed into the class of u+v in the resulting partition. Thus,

$$P_F = \{u,20|v,30|u+v,50\}.$$

The analysis continues as before up to node H, resulting in

$$P_H = \{u,20|v,30|u+v,50|x,10|y,40\}.$$

In the case of the f_1 optimizing function, the expression x+y at node H is placed into a distinct class. The operands x and y, however, are propagated constants since they are equivalent to 10 and 40, respectively. The expression x+y is equivalent to 50 which is already in the partition. Thus, x+y is added to the class of 50, resulting in

$$P_I = \{u,20|v,30|u+v,50,x+y|x,10|y,40\}.$$

Similarly, the output pool from node I is

$$\{u,20|v,30,y-x|u+v,50,x+y|x,10|y,40\}.$$

The analysis above depends upon the ability to recognize certain expressions as constants and the ability to compute the constant value of an expression when the operands are all propagated constants. It is also implicit that no two differing constants are in the same class.

An optimizing function f_2 which combines constant propagation with common subexpression elimination can be constructed from f_1 by altering step (3) as follows:

3a. create a new class in P containing e and add all (partial) computations which occur in the program graph and which have operands equivalent to those of e (structure the pool as before).

3b. If e does not evaluate to a constant value based upon propagated constant operands, then no further processing is required (same as step (3) of f_1); otherwise let z be the constant value of e. If z is already in the partition P then combine the class of z with the class of e in the resulting partition. If z is not in the partition P, then add z to the class of e. The expression e becomes a propagated constant in either case.

The function f_2 is stated formally and its properties are investigated elsewhere [33].

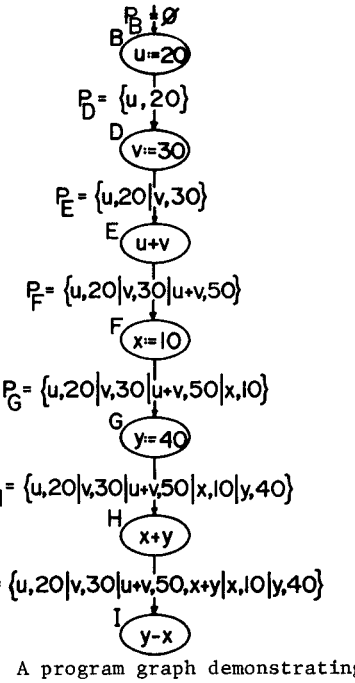

Figure 3. A program graph demonstrating the effects of constant propagation.

6. EXPRESSION OPTIMIZATION

Expression optimization, as defined earlier, includes common subexpression elimination, constant propagation, and register optimization. The first two forms of optimization are covered by the f_2 optimizing function; only register optimization needs to be considered. It will be shown below that f_2 also provides a simple form of register optimization.

In general, global register optimization involves the assignment of high speed registers (accumulators and index registers) throughout a program in such a manner that the number of store-fetch sequences between the high-speed registers and central memory is minimized. The store-fetch sequences arise in two ways. The first form involves redundant fetches from memory. Consider the sequence of expressions

a:=b+c; d:=a+e;

for example. A straight-forward translation of these statements for a machine with multiple general-purpose registers might be

r_1:=b; r_2:=c; r_1:=r_1+r_2; a:=r_1;

r_1:=a; r_2:=e; r_1:=r_1+r_2; d:=r_1.

Note, however, that the operation r_1:=a is not necessary since r_1 contains the value of the variable a before the operation. McKeeman [38] discusses a technique called "peephole optimization" which eliminates these redundant fetches within a basic block.

Figure 4 shows a program corresponding to the register operations above. The f_2 optimizing function is applied to each successive node in the graph, resulting in the optimizing pools shown in the Figure. In particular, note that

$$P_E = \{a, r_1 | b | r_2, c\}.$$

The operation at node E assigns the variable a to the register r_1. Since a is already in the class of r_1, however, the operation is redundant and can be eliminated. Hence, the f_2 optimizing function can be used to generalize peephole optimization. Further, the algorithm A extends f_2 to allow global elimination of redundant register load operations.

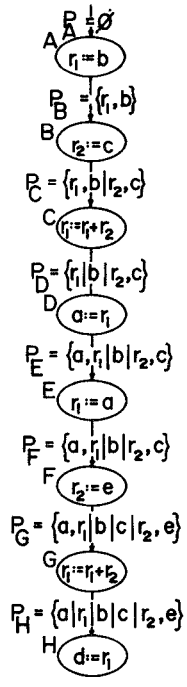

Figure 4. Elimination of redundant register load operations.

The second source of store-fetch sequences arises when registers are in use and must be released temporarily for another purpose. The contents of the busy register is stored into a central memory location and restored again at a later point in the program. An optimal register allocation scheme would minimize the number of temporary stores. This form of register optimization has been treated on a local basis, including algorithms which arrange arithmetic computations in order to reduce the total number of registers required in the evaluation [5,27,36,39,43,45,46]. Global register allocation has also been formulated as an integer programming problem by Day [14], given that register interference and cost of data displacement from registers is known. No complete solution to the global register allocation problem is known by the author at this time.

A solution to the global register allocation problem will be aided by the analysis of "live" and "dead" variables at each node in the program graph. A variable v is live at a node N if v could possibly be referenced in an expression subsequent to node N. The variable v is dead otherwise. Recent work has been done by Kennedy [32] using interval analysis techniques to detect live and dead variables on a global basis.

An optimizing function f_3 can be constructed which produces a set of live expressions at each node in the graph. The detection of live expressions requires the analysis to proceed from the end of the program toward the beginning. Figure 5 shows the graph of Figure 4 with the direction of the edges reversed. The live expressions at the beginning of the graph correspond to the live expressions at the end of program execution; hence, $P_H = \emptyset$ (there are no live expressions at the end of execution). The expression d:=r_1 at node H refers to the expression r_1. Thus, r_1 is live ahead of node H. This fact is recorded by including r_1 in P_G,

$$P_G = \{r_1\}.$$

Since r_1 is assigned a new value at node G, it becomes a dead expression, but, since r_1 is also involved in the expression r_1+r_2, it immediately becomes a live expression again. Thus,

$$P_F = \{r_1, r_2, r_1+r_2\}.$$

The analysis continues, producing the optimizing pools associated with each node in Figure 5. The expressions which are live at node C, for example, are

$$P_B = \{e, r_1, r_2, r_1+r_2\}.$$

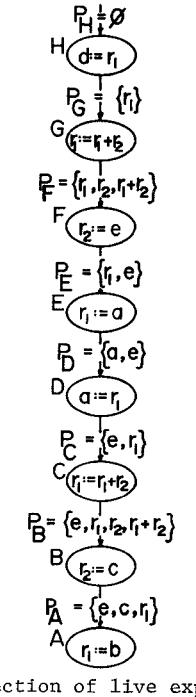

Figure 5. Detection of live expressions in a reversed program graph.

The optimizing function $f_3(N,P)$ which provides live expression analysis can be informally stated as follows:

1. If the expression at node N involves an assignment to a variable, let d be the destination of the assignment; set $P \leftarrow P - \{e \mid d$ is a subexpression in $e\}$(d and all expressions containing d become dead expressions).
2. Consider each partial computation e at node N. Set $P \leftarrow P \cup \{e\}$ (e becomes a live expression). The value of $f_3(N,P)$ is the altered value of P.

The algorithm A can then be applied to the reversed program graph using the optimizing function f_3. The exit nodes of the original graph become the entry nodes of the reversed graph. In addition, the meet operation of the algorithm A is the set union operation $\cup$. The union operation induces the partial ordering given by

$$P_1 \leq P_2 \iff P_1 \cup P_2 = P_1 \iff P_1 \supseteq P_2, \ \forall P_1, P_2 \in \underline{P}',$$

where $\underline{P}$ is the set of (partial) computations which occur in the program graph. Note that $\underline{0} = \underline{P}'$ and $\underline{1} = \emptyset$ in this case. Thus, all initial approximate pools in the algorithm A are set to $\emptyset$.

There is a simple generalization of detection of live expressions to "minimum distance analysis" where each live expression is accompanied by the minimum distance to an occurrence of the expression. The optimizing pools in this case are sets of ordered pairs (e,d), where e is a live expression and d is the minimum distance (in program steps) to an occurrence of e. The optimizing function extends live expression analysis by tabulating a distance measure as the live expression analysis proceeds. In addition, the meet operation consists of both set union and a comparison of the distances corresponding to each live expression. This minimum distance information can then be used in the register replacement decision: whenever all registers are busy and contain live expressions, the register containing the live expression with the largest distance to its occurrence is displaced.

Examples are given in the section which follows demonstrating the f_2 and f_3 optimizing functions when used in conjunction with the algorithm A.

7. A TABULAR FORM FOR THE ALGORITHM A

The processing of the algorithm A can be expressed in a tabular form. The tabular form allows presentation of a number of examples, and provides an intuitive basis for implementing the optimizing techniques. In particular, this form allows representation of the approximate optimizing pools at each node, the elements of L, and the node traversing decision. As shown in Table I, the column labeled "N" contains the current node being processed (i.e., the N in $L' = (N,P_i)$ in step A5). The column labeled "$P_N \leftarrow P_N \wedge P_i$" shows the change in the approximate pool at node N when the node is traversed in step A5. The column marked "$f(N,P_N)$" contains the output optimizing pool produced by traversing the node N (the set braces are omitted for convenience of notation). The last column, marked "L," represents the set of nodes remaining to be processed (the set L of the algorithm A).

Paraphrasing the algorithm A, the tabular form is processed as follows.

1. List all entry nodes and entry pools vertically in the right-hand columns, with entry node e_i in column L, and associated entry pool x_i in column $f(N,P_N)$. Normally, there is only one entry node, with the null set as an entry pool.
2. Select an L' from L as follows. Choose any node from column L, say node N. If there are no elements remaining in L then the algorithm halts. The line where N was added to L contains the associated output pool P_i in the column $f(N,P_N)$. Eliminate L' from L by crossing out N from column L.
3. Using $L' = (N,P_i)$ from step 2, scan the table from the bottom upward to the first occurrence of node N in column N. The current approximate pool P_N is adjacent in the column $P_N \leftarrow P_N \wedge P_i$. If node N has not appeared in column N, then assume the first approximation to $P_N = \underline{1}$ (and hence, $P_N \leftarrow \underline{1} \wedge P_i = P_i$).
4. If $P_N \leq P_i$ then go to step 2. Otherwise, write the node name N in column N and the value of the new approximate pool determined by $P_N \wedge P_i$ in the column marked $P_N \leftarrow P_N \wedge P_i$. Compute the output pool based upon the new approximate pool P_N in the column $f(N,P_N)$, and write the names of the immediate successors of N in column L. Go back to step 2.

Upon termination of this algorithm, the table is scanned from bottom to top; the first occurrence of each node $N \in \underline{N}$ is circled. The pool associated with each circled node in column $P_N \leftarrow P_N \wedge P_i$ is the final pool for that node. Any nodes of $\underline{N}$ which do not appear in column N cannot be reached from an entry node, and can be eliminated from the program graph.

Table I shows the analysis of the program graph given in Figure 1, using the f_2 optimizing function. The entry node set for this analysis is $\underline{e} = \{(A,\emptyset)\}$, as before. L is treated as a stack; elements are removed from the lower right position of column L in step 2. After processing the graph, the final pools at each node are listed in the table opposite the circled nodes. The final pool at node E, for example, is

$$P_E = \{a,1 \mid b,2 \mid d,a+b,3\}.$$

The final pools determined by the algorithm correspond to those determined previously in Section 2.

TABLE I

step	N	$P_N \leftarrow P_N \wedge P_i$	$f(N,P_N)$	L
1			$\emptyset$	A
2	Ⓐ	$\emptyset$	a,1	B
3	Ⓑ	a,1	a,1 | c,0	C
4	C	a,1 | c,0	a,1 | c,0 | b,2	D
5	D	a,1 | c,0 | b,2	a,1 | c,0 | b,2 | d,a+b,3	E
6	E	a,1 | c,0 | b,2 | d,a+b,3	a,1 | c,0 | b,2 | e,b+c | d,a+b,3	F
7	F	a,1 | c,0 | b,2 | e,b+c | d,a+b,3	a,1 | b,2 | e | d,a+b,3 | c,4	C
8	Ⓒ	a,1	a,1 | b,2	D
9	Ⓓ	a,1 | b,2	a,1 | b,2	E
10	Ⓔ	a,1 | b,2 | d,a+b,3	a,1 | b,2 | d,a+b,3 | b+c,e	F
11	Ⓕ	a,1 | b,2	a,1 | b,2 | c,4	C

Figure 6 shows a program graph with two parallel feedback loops. The analysis of this program graph is given in Table II, using the f_2 optimizing function. Note that in step (8),

$$P_F = \{10|y|x,5,u\}.$$

Applying $f_2(F,P_F)$, the resulting output pool is

$$\{10|y|x,5,u|u\cdot y,x\cdot y\}.$$

The expression x·y is placed into the class of u·y when the partition is structured. That is, x·y is an expression which occurs in the program, and x·y is operand equivalent to u·y. Thus, x·y must be added to the class of u·y in the output pool. The redundant expression x·y is detected at node G since the final pool P_G contains x·y.

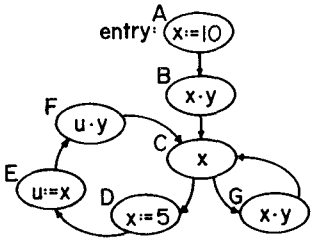

Figure 6. A program graph with two parallel feedback loops.

TABLE II

step	N	$P_N \leftarrow P_N \wedge P_i$	f(N,P_N)	L
1			$\emptyset$	$\not\emptyset$
2	Ⓐ	$\emptyset$	x,10	$\not\emptyset$
3	Ⓑ	x,10	x,10|y|x·y	$\emptyset$
4	C	x,10|y|x·y	x,10|y|x·y	$\not\emptyset,\emptyset$
5	G	x,10|y|x·y	x,10|y|x·y	$\emptyset$
6	D	x,10|y|x·y	10|y|x·y|x,5	$\not\emptyset$
7	Ⓔ	10|y|x,5	10|y|x,5,u	$\not\emptyset$
8	Ⓕ	10|y|x,5,u	10|y|x,5,u|u·y,x·y	$\emptyset$
9	Ⓒ	x|10|y|x·y	x|10|y|x·y	$\not\emptyset,\emptyset$
10	Ⓖ	x|10|y|x·y	x|10|y|x·y	$\emptyset$
11	Ⓓ	x|10|y|x·y	x,5|10|y	$\not\emptyset$

Global live expression analysis can be performed on the program graph of Figure 6 by reversing the graph, as shown in Figure 7. Given that node C is the exit node of the original graph, node C becomes the entry node of the reversed graph. Thus, $\underline{\mathcal{C}} = \{(C,\emptyset)\}$ in the analysis shown in Table III, using the f_3 optimizing function. For example, the final pool

$$P_A = \{x,y,x\cdot y\}$$

indicates that the expressions x, y, and x·y are live immediately following node A in the original graph.

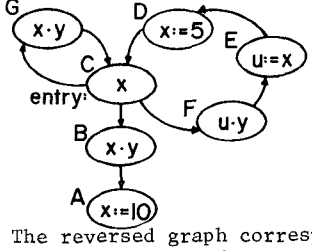

Figure 7. The reversed graph corresponding to the program graph of Figure 6.

TABLE III

step	N	$P_N \leftarrow P_N \wedge P_i$	f(N,P_N)	L
1			$\emptyset$	$\not\emptyset$
2	C	$\emptyset$	x	$\not\emptyset,\not\emptyset,\not\emptyset$
3	G	x	x,y,x·y	$\not\emptyset$
4	Ⓒ	x,y,x·y	x,y,x·y	$\not\emptyset,\not\emptyset,\not\emptyset$
5	Ⓖ	x,y,x·y	x,y,x·y	$\not\emptyset$
6	Ⓕ	x,y,x·y	x,y,x·y,u,u·y	$\not\emptyset$
7	Ⓔ	x,y,x·y,u,u·y	x,y,x·y	$\not\emptyset$
8	Ⓓ	x,y,x·y	y	$\not\emptyset$
9	Ⓑ	x,y,x·y	x,y,x·y	$\not\emptyset$
10	Ⓐ	x,y,x·y	y,10	

This tabular form can be used for processing any program graph using an optimizing function which satisfies the conditions of the algorithm A.

8. IMPLEMENTATION NOTES

Implementation of the above optimizing techniques in a practical compiler is considered below. In particular, the optimizer operates upon an intermediate form of the program, such as tree structures or Polish [24], augmented by branching information. The control flow analyzer accepts the intermediate form and calls the various optimizing functions to process each basic block, roughly paralleling the tabular form given previously. A single stack can be used to list uninvestigated basic blocks, corresponding to "L" of the tabular form. Pool information must be maintained for each basic block corresponding to the "$P_N \leftarrow P_N \wedge P_i$" column, but may be discarded and replaced if the node is encountered again in the analysis (i.e., the node reappears in column "N"). The output optimizing pools found in column "f(N,P_N)," however, can be intersected with all immediate successors as they are produced, and thus need not be maintained during analysis. The final optimizing pools (determined by "scanning" the tabular form) are simply the current pools attached to each basic block.

The optimizing functions and corresponding meet operations are generally simple to implement using bit strings for sets, and lists for ordered pairs. Common subexpression elimination, however, requires further consideration since direct representation and manipulation of structured partitions is particularly unwieldy.

One approach to handling structured partitions allows direct representation of the classes, but limits the number of expressions which appear. A list of all (sub)expressions is constructed by prescanning the program (an optimizing function which always returns $\emptyset$ is useful for this scan). When a partition is structured, only those expressions which occur in the expression list are included. The set of eligible expressions can be further reduced by first performing live expression analysis. The expressions which appear in a partition are limited to the live expressions at the point the partition is generated. The use of live expression analysis before common subexpression elimination will generally reduce partition size and improve the convergence rate of the analysis algorithm.

A second approach to representation of structured partitions involves the assignment of "value

numbers" to the expressions in the optimizing pools [13,24,33,34]. A value number is a unique integer assigned to all elements of the same class. The sequence of statements

a:=b+c; d:=b; e:=a;

results in the structured partition

$$P_1 = \{ b,d \mid c \mid b+c,d+c,a,e \}.$$

Next, assign the value numbers 1, 2, and 3 to the three classes, and replace the expressions b+c and d+c by (1)+(2), representing the addition of elements of class (1) and class (2). P_1 can now be written as

$$P_2 = \{ \underset{(1)}{b,d} \mid \underset{(2)}{c} \mid \underset{(3)}{(1)+(2),a,e} \}.$$

Similarly, the sequence of assignments

a:=d; b:=c; e:=b+c;

produces the structured partition represented by

$$P_2 = \{ \underset{(4)}{a,d} \mid \underset{(5)}{b,c} \mid \underset{(6)}{(5)+(5),e} \}.$$

which expands to

$$P_2 = \{a,d \mid b,c \mid b+c,b+b,c+b,c+c,e\}$$

Thus, the assignment of value numbers provides a data structure whose size is linear in the number of expressions in the basic block. In addition, the value number representation is particularly easy to construct and use in the detection of common subexpressions.

Given two partitions P_1 and P_2 in value number form, the meet operation $P = P_1 \wedge P_2$ can be iteratively computed. The computation proceeds as follows. Construct a list C consisting of the number of occurrences of each value number in P_1. The elements of C thus provide a count of the number of elements in each class of P_1. This count is decremented whenever an element of the class is processed, until the count goes to zero indicating the entire class is exhausted.

A list R is also maintained which gives a mapping of the class numbers in P_1 and P_2 to the resulting class numbers in P. The elements of R are of the form $r(r_1,r_2)$, indicating that value number r_1 from P_1 and value number r_2 from P_2 map to value number r in the resulting partition P. R is built during the construction of P.

The elements of P_1 are scanned and processed until the classes of P_1 are exhausted. Suppose q is an identifier in P_1 with value number v_1. The count corresponding to v_1 in the list C is first decremented. If q does not occur in P_2 then the next element of P_1 is selected. Otherwise, let v_2 be the value number corresponding to q in P_2. R is scanned for an element $v(v_1,v_2)$; if not found, a new value number v is assigned, and $v(v_1,v_2)$ is added to R. The identifier q is placed into P with value number v.

If the element selected from P_1 is not an identifier, then it is an expression of the form $(n_1)\ \theta\ (m_1)$ with value number v_1, where n_1 and m_1 are value numbers in P_1 (assuming all operations θ are binary). If the count of either class (n_1) or (m_1) is non-zero in C, defray the processing of this expression; otherwise, decrement the count for class (v_1) in C, as above. Examine R for pairs of elements $n(n_1,n_2)$ and $m(m_1,m_2)$ where n_2 and m_2 are value numbers in P_2. For each such pair, search P_2 for an entry $(n_2)\ \theta\ (m_2)$. If found, let v_2 be the value number of this matched expression. Scan R for an element of the form $v(v_1,v_2)$, and make a new entry if not found, as above. The expression $(n)\ \theta\ (m)$ with value number v is then placed into the intersection P.

As an example, consider the class intersection of the partitions P_1 and P_2 given previously. These partitions are represented by the value number tables

P_1 exp	val#
b	(1)
d	(1)
c	(2)
(1)+(2)	(3)
a	(3)
e	(3)

P_2 exp	val#
a	(4)
d	(4)
b	(5)
c	(5)
(5)+(5)	(6)
e	(6)

The class count list C for the partition P_1 is initially

val#	count
(1)	2
(2)	1
(3)	3

The identifiers b, d, and c are processed first, reducing the class counts for (1) and (2) to zero in C. The class mapping list at this point is

$$R = \{7(1,5),\ 8(1,4),\ 9(2,5)\}$$

The identifiers b, d, and c are placed into P with value numbers 7, 8, and 9, respectively. The expression (1)+(2) with value number (3) is then processed from P_1, since the class counts for both (1) and (2) are zero. Based upon the mappings in R, P_2 is searched for an occurrence of (5)+(5) or (4)+(5). Since (5)+(5) occurs in P_2 with value number (6), R is scanned for an element of the form v(3,6), and, since no such element is found, 10(3,6) is added to R. The expression (7)+(9) with value number (10) is included in P. The identifier a is then processed, resulting in another mapping 11(3,4) in R; a is added to P with value number (11). Finally, the identifier e from P_1 with value number (3) is processed. A match is found in P_2 with value number (6). Since the element 10(3,6) is already in R, e is added to P with value number (10). The final value of the class list is

$$R = \{7(1,5),\ 8(1,4),\ 9(2,5),\ 10(3,6),\ 11(3,4)\}$$

which can now be discarded. The value of the resulting partition P is

exp	val#
b	(7)
d	(8)
c	(9)
(7)+(9)	(10)
a	(11)
e	(10)

which represents the structured partition

$$\{b \mid d \mid c \mid b+c,e \mid a\}$$

Note that the predicate $P_2 \geq P_1$ is easily computed

during this process.

The control flow analysis algorithm has been implemented as a general-purpose optimizing module, including several optimizing functions. The implementation is described in some detail elsewhere [33].

9. CONCLUSIONS

An algorithm has been presented which, in conjunction with various optimizing functions, provides global program optimization. Optimizing functions have been described which provide constant propagation, common subexpression elimination, and a degree of register optimization.

The functions which have been given by no means exhaust those which are useful for optimization. Simplifying formal identities such as 0+x = 0+x = x can be incorporated to further coalesce equivalence classes at each application of the f_2 optimizing function. In addition, it may be possible to develop functions which extend live expression analysis to completely solve the global register allocation problem.

REFERENCES

1. Aho, A., Sethi, R., and Ullman, J. A formal approach to code optimization. Proceedings of a Symposium on Compiler Optimization. University of Illinois at Urbana-Champaign, July, 1970.
2. Allen, F. Program optimization. In Annual Review in Automatic Programming, Pergamon Press, 5(1969), 239-307.
3. --- A basis for program optimization. IFIP Congress 71, Ljubljana, August, 1971, 64-68.
4. --- Control flow analysis. Proceedings of a Symposium on Compiler Optimization, University of Illinois at Urbana-Champaign, July, 1970.
5. Anderson, J. A note on some compiling algorithms. Comm. ACM 7, 3 (March 1964), 149-150.
6. Arden, B. Galler, B., and Graham, R. An algorithm for translating boolean expressions. Jour. ACM 9, 2(April 1962), 222-239.
7. Bachmann, P. A contribution to the problem of the optimization of programs. IFIP Congress 71, Ljubljana, August, 1971, 74-78.
8. Ballard, A., and Tsichritzis, D. Transformations of programs. IFIP Congress 71, Ljubljana, August, 1971, 89-93.
9. Breuer, M. Generation of optimal code for expressions via factorization. Comm. ACM 12, 6(June 1970), 333-340.
10. Busam, V., and Englund, D. Optimization of expressions in FORTRAN. Comm. ACM 12, 12(Dec. 1969), 666-674.
11. Cocke, J. Global common subexpression elimination. Proceedings of a Sybposium on Compiler Optimization. University of Illinois at Urbana-Champaign, July, 1970.
12. ---, and Miller, R. Some analysis techniques for optimizing computer programs. Proc. Second International Conference of System Sciences, Hawaii, January, 1969, 143-146.
13. ---, and Schwartz, J. Programming Languages and Their Compilers: Preliminary Notes. Courant Institute of Mathematical Sciences, New York University, 1970.
14. Day, W. Compiler assignment of data items to registers. IBM Systems Journal, 8, 4(1970), 281-317.
15. Earnest, C., Balke, K., and Anderson, J. Analysis of graphs by ordering nodes. Jour. ACM 19, 1(Jan. 1972), 23-42.
16. Elson, M., and Rake, S. Code generation technique for large language compilers. IBM Systems Journal 3(1970), 166-188.
17. Fateman, R. Optimal code for serial and parallel computation. Comm. ACM 12, 12(Dec. 1969), 694-695.
18. Finkelstein, M. A compiler optimization technique. The Computer Review (Feb. 1968), 22-25.
19. Floyd, R. An algorithm for coding efficient arithmetic operations. Comm. ACM 4, 1(Jan. 1961), 42-51.
20. Frailey, D. Expression Optimization using unary complement operators. Proceedings of a Symposium on Compiler Optimization, University of Illinois at Urbana-Champaign, July, 1970.
21. ---, A study of optimization using a general purpose optimizer. (PhD Thesis) Purdue University, Lafayette, Ind., January 1971.
22. Freiburghouse, R. The MULTICS PL/I compiler. AFIPS Conf. Proc. FJCC (1969), 187-199.
23. Gear, C. High speed compilation of efficient object code. Comm. ACM 8, 8(Aug. 1965), 483-488.
24. Gries, D. Compiler Construction for Digital Computers. John Wiley and Sons, Inc., New York, 1971.
25. Hill, V., Langmaack, H., Schwartz, H., and Seegumuller, G. Efficient handling of subscripted variables in ALGOL-60 compilers. Proc. Symbolic Languages in Data Processing, Gordon and Breach, New York, 1962, 331-340.
26. Hopkins, M. An optimizing compiler design. IFIP Congress 71, Ljubljana, August, 1971, 69-73.
27. Horowitz, L., Karp, R., Miller, R., and Winograd, S. Index register allocation. Jour. ACM 13, 1(Jan. 1966), 43-61.
28. Huskey, H., and Wattenberg, W. Compiling techniques for boolean expressions and conditional statements in ALGOL-60. Comm. ACM 4, 1(Jan. 1961), 70-75.
29. Huskey, H. Compiling techniques for algebraic expressions. Computer Journal 4, 4(April 1961), 10-19.

30. Huxtable, D. On writing an optimizing translator for ALGOL-60. In Introduction to System Programming, Academic Press, Inc., New York, 1964.

31. IBM System/360 Operating System, FORTRAN IV (G and H) Programmer's Guide. C28-6817-1, International Business Machines, 1967, 174-179.

32. Kennedy, K. A global flow analysis algorithm. Intern. J. of Computer Mathematics, Section A, Vol. 3, 1971, 5-15.

33. Kildall, G. Global expression optimization during compilation. Technical Report No. TR# 72-06-02, University of Washington Computer Science Group, University of Washington, Seattle, Washington, June, 1972.

34. --- A code synthesis filter for basic block optimization. Technical Report No. TR# 72-01-01, University of Washington Computer Science Group, University of Washington, Seattle, Washington, January, 1972.

35. Lowry, E., and Medlock, C. Object code optimization. Comm. ACM 12, 1(Jan. 1969), 13-22.

36. Luccio, F. A comment on index register allocation. Comm. ACM 10,9 (Sept. 1967), 572-572-574.

37. Maurer, W. Programming-An Introduction to Computer Language Technique. Holden-Day, San Francisco, 1968, 202-203.

38. McKeeman, W. Peephole optimization. Comm. ACM 8, 7(July 1965), 443-444.

39. Nakata, I. On compiling algorithms for arithmetic expressions. Comm. ACM 19, 8(Aug. 1967), 492-494.

40. Nievergelt, J. On the automatic simplification of computer programs. Comm. ACM 8, 6(June 1965), 366-370.

41. Painter, J. Compiler effectiveness. Proceedings of a Symposium on Compiler Optimization, University of Illinois at Urbana-Champaign, July, 1970.

42. Randell, B., and Russell, L. ALGOL 60 Implementation. Academic Press, Inc., New York, 1964.

43. Redziejowski, R. On arithmetic expressions and trees. Comm. ACM 12, 2(Feb. 1969), 81-84.

44. Ryan, J. A direction-independent algorithm for determining the forward and backward compute points for a term or subscript during compilation. Computer Journal 9, 2(Aug. 1966), 157-160.

45. Schnieder, V. On the number of registers needed to evaluate arithmetic expressions. BIT 11(1971), 84-93.

46. Sethi, R., and Ullman, J. The generation of optimal code for arithmetic expressions. Jour. ACM 17, 4(Oct. 1970), 715-728.

47. Wagner, R. Some techniques for algebraic optimization with application to matrix arithmetic expressions. Thesis, Carnegie-Mellon University, June, 1968.

48. Yershov, A. On programming of arithmetic operations. Comm. ACM 1, 8(Aug. 1958), 3-6.

49. --- ALPHA-an automatic programming system of high efficiency. Jour. ACM 13, 1(Jan. 1966), 17-24.

APPENDIX A

1 A1: $L = \{(A,\emptyset)\}$

2 A3: $L' = (A,\emptyset)$, $L = \emptyset$

3 A4: $P_N = P_A = \underline{1}$, $P_i = \emptyset$, $P_A \not\leq P_i$, $P_A \leftarrow P_A \wedge P_i = P_i = \emptyset$

4 A5: $P_A = \emptyset$, $L = \{(B,\{(a,1)\})\}$

5 A3: $L' = (B,\{(a,1)\})$, $L = \emptyset$

6 A5: $P_B = \{(a,1)\}$, $L = \{(C,\{(a,1),(c,0)\})\}$

7 A3: $L' = (C,\{(a,1),(c,0)\})$, $L = \emptyset$

8 A5: $P_C = \{(a,1),(c,0)\}$, $L = \{(D,\{(a,1),(c,0),(b,2)\})\}$

9 A3: $L' = (D,\{(a,1),(c,0),(b,2)\})$, $L = \emptyset$

10 A5: $P_D = \{(a,1),(c,0),(b,2)\}$, $L = \{(E,\{(a,1),(c,0),(b,2),(d,3)\})\}$

11 A3: $L' = (E,\{(a,1),(c,0),(b,2),(d,3)\})$, $L = \emptyset$

12 A5: $P_E = \{(a,1),(c,0),(b,2),(d,3)\}$, $L = \{(F,\{(a,1),(c,0),(b,2),(d,3),(e,2)\})\}$

13 A3: $L' = (F,\{(a,1),(c,0),(b,2),(d,3),(e,2)\})$, $L = \emptyset$

14 A5: $P_F = \{(a,1),(c,0),(b,2),(d,3),(e,2)\}$, $L = \{(C,\{(a,1),(c,4),(b,2),(d,3),(e,2)\})\}$

15 A3: $L' = (C,\{(a,1),(c,4),(b,2),(d,3),(e,2)\})$, $L = \emptyset$

16 A5: $P_C = \{(a,1)\}$, $L = \{(D,\{(a,1),(b,2)\})\}$

17 A3: $L' = (D,\{(a,1),(b,2)\})$, $L = \emptyset$

18 A5: $P_D = \{(a,1),(b,2)\}$, $L = \{(E,\{(a,1),(b,2),(d,3)\})\}$

19 A3: $L' = (E,\{(a,1),(b,2),(d,3)\})$, $L = \emptyset$

20 A5: $P_E = \{(a,1),(b,2),(d,3)\}$, $L = \{(F,\{(a,1),(b,2),(d,3)\})\}$

21 A3: $L' = (F,\{(a,1),(b,2),(d,3)\})$, $L = \emptyset$

22 A5: $P_F = \{(a,1),(b,2),(d,3)\}$, $L = \{(C,\{(a,1),(b,2),(d,3),(c,4)\})\}$

23 A3: $L' = (C,\{(a,1),(b,2),(d,3),(c,4)\})$, halt.

APPENDIX B

The proof of Theorem 2 is given below. First note that given a program graph G with multiple entry nodes, an augmented graph G' can be constructed with only one entry node with entry pool $\underline{0}$. The construction is as follows. Let $\mathcal{E} = \{e_1,e_2,\ldots e_k\}$ be the entry node set and $\underline{\mathcal{E}} = \{(e_1,x_1),(e_2,x_2),\ldots,(e_k,x_k)\}$ be the entry pool set corresponding to a particular analysis. Consider the augmented graph $G' = \langle \underline{N}', \underline{E}' \rangle$ where

$\underline{N}' = \underline{N} \cup \{v,v_1,\ldots,v_k\} \ni v,v_i \notin \underline{N}\ \forall i,\ 1 \leq i \leq k$, and

$$\underline{E}' = \underline{E} \cup$$
$$\{(v,v_1),(v,v_2),\ldots,(v,v_k),(v_1,e_1),..,(v_k,e_k)\}.$$

The augmented graph G' has a single entry node v and entry node set $\mathcal{E}' = \{v\}$. The functional value of f is defined for these nodes as

$$f(v,P) = \underline{0} \quad \forall P \in \underline{P}, \text{ and}$$

$$f(v_i,P) = x_i \quad \forall P \in \underline{P}, \ 1 \le i \le k.$$

Hence, the analysis proceeds as if there is only a single entry node with entry pool $\underline{0}$; i.e., $\mathcal{E}' = \{(v,\underline{0})\}$.

Lemma 1. If $f(N,P_1 \wedge P_2) = f(N,P_1) \wedge f(N,P_2)$ then $P_1 \le P_2 \Rightarrow f(N,P_1) \le f(N,P_2)$, $\forall N \in \underline{N}$, $P_1,P_2 \in \underline{P}$.

Proof. The proof is immediate since $P_1 \le P_2 \Rightarrow f(N,P_1 \wedge P_2) = f(N,P_1) = (f(N,P_1) \wedge f(N,P_2)) \Rightarrow f(N,P_1) \le f(N,P_2)$ ●

Lemma 2. Let $X \subseteq \underline{P}$, if $f(N,P_1 \wedge P_2) = f(N,P_1) \wedge f(N,P_2)$ $\forall N \in \underline{N}$, $P_1,P_2 \in \underline{P}$ then

$$f(N, \bigwedge_{x \in X} x) = \bigwedge_{x \in X} f(N,x).$$

Proof. The proof proceeds by induction on the cardinality of X, denoted by $\mathcal{C}(X)$. If $\mathcal{C}(X) = 1$ then $f(N, \bigwedge_{x \in X} x) = f(N,x)$ and the lemma is trivially true. If $\mathcal{C}(X) = k$, $k>1$, assume lemma is true for all $X' \ni \mathcal{C}(X') < k$. Let $y \in X$ and $X' = X - \{y\}$.

$$f(N, \bigwedge_{x \in X} x) = f(N, y \wedge (\bigwedge_{x \in X'} x)) = f(N,y) \wedge f(N, \bigwedge_{x \in X'} x) =$$
$$f(N,y) \wedge (\bigwedge_{x \in X'} f(N,x)) = \bigwedge_{x \in X} f(N,x)$$ ●

Proof of Theorem 2. It will first be shown by induction on the path length that

$$P_N \le X_N \quad \forall N \in \underline{N}.$$

Consider the following proposition on n:

$P_N \le f(p_n,f(p_{n-1},\ldots,f(p_1,\underline{0}))\ldots)$ for all final pools P_N and paths of length n from the entry node p_1 with entry pool $\underline{0}$ to node N, $\forall N \in \underline{N}$.

The trivial case is easily proved. The only node which can be reached by a path of length 0 from the entry node p_1 is p_1 itself. Hence, it is only necessary to show that $P_{p_1} \le \underline{0}$. This is immediate, however, since $(p_1,\underline{0})$ is initially placed into L in step A1, and extracted in step A3 as $L' = (p_1,\underline{0})$. But, P_{p_1} is initially $\underline{1}$, and hence $P_{p_1} \not\le P_i = \underline{0}$ in step A4. Thus, $P_{p_1} \leftarrow P_{p_1} \wedge \underline{0} = \underline{0}$ in step A5. Thus, it follows that $P_{p_1} = \underline{0} \le \underline{0}$.

Suppose the proposition is true for all $n<k$, for $k>0$. That is, $P_N \le f(p_n,\ldots,f(p_1,\underline{0}))\ldots)$ for all paths of length less than k from p_1 to node N, for each node $N \in \underline{N}$.

Let $K \in \underline{N} \ni$ a path $(P_1,\ldots,p_k,K)$ of length k. It will be shown that $P_K \le f(p_k,f(p_{k-1},\ldots,f(p_1,\underline{0}))\ldots)$.

Consider each immediate predecessor in $I^{-1}(K)$. Let p_k denote one such predecessor, and let $T = f(p_{k-1},\ldots,f(p_1,\underline{0}))\ldots)$. By inductive hypothesis, $P_{p_k} \le T$. It will be shown that $P_K \le f(p_k,T)$.

Since P_{p_k} is the final approximation to the pool at p_k, $(K,f(p_k,P_{p_k}))$ must have been added to L in step A5. But, $P_{p_k} \le T \Rightarrow f(p_k,P_{p_k}) \le f(p_k,T)$ by Lemma 1. The pair $(K,f(p_k,P_{p_k}))$ must be processed in step A3 before the algorithm halts. Thus, either $P_K \le f(p_k,P_{p_k})$ in step A4, or $P_K \leftarrow P_K \wedge f(p_k,P_{p_k})$. In either case, $P_K \le f(p_k,P_{p_k})$. But,

$$P_K \le f(p_k,P_{p_k}) \le f(p_k,T) \Rightarrow P_K \le f(p_k,T)$$
$$\Rightarrow P_K \le f(p_k,f(p_{k-1},\ldots,f(p_1,\underline{0}))\ldots).$$

Thus, since the proposition holds for paths of length k, it follows by induction that the proposition is true for all paths from the entry node to node N, for all $N \in \underline{N}$.

The following claim will be proved in order to show that $X_N \le P_N$ for all $N \in \underline{N}$: at any point in the processing of G by the algorithm A, either N has not been encountered in step A5, or $X_N \le P_N$, where P_N is the current approximate pool associated with node N, for all $N \in \underline{N}$. The proof proceeds by induction on the number of times step A5 has been executed. Suppose step A5 has been executed only once. Then $L' = (p_1,\underline{0})$ and the only node encountered in step A5 is the entry node p_1. The entry pool $\underline{0}$ corresponds to a path of length zero from p_1 to p_1. Thus, $\underline{0} \in F_{p_1} \Rightarrow X_{p_1} = \underline{0}$ and the proposition is trivially true since $X_{p_1} = \underline{0} \le P_{p_1} = \underline{0}$.

Suppose that either N has not been encountered in step A5, or $X_N \le P_N$ $\forall N \in \underline{N}$ when step A5 has been executed $n<k$ times, $k>1$. Consider the k^{th} execution of step A5. Let $L' = (N,T)$ where $T = f(N',P_{N'})$ for some $N' \in I^{-1}(N)$. The pair (N,T) was added to L when the node N' was processed in the n^{th} execution of step A5, for $n<k$. Hence, $X_{N'} \le P_{N'}$ by inductive hypothesis. But, using Lemma 2,

$$X_N \le \bigwedge_{\substack{\forall \text{ paths} \\ (p_1,\ldots,p_t,N',N)}} f(N',f(p_t,\ldots,f(p_1,\underline{0}))\ldots) =$$
$$f(N', \bigwedge_{\substack{\forall \text{ paths} \\ (p_1,\ldots,p_t,N')}} f(p_t,f(p_{t-1},\ldots,f(p_1,\underline{0}))\ldots)$$
$$= f(N',X_{N'}).$$

$X_{N'} \le P_{N'}$ and thus $X_N \le f(N',X_{N'}) \Rightarrow$
$X_N \le f(N',P_{N'}) = T$, using Lemma 1.

If this step is the first occurrence of node N in A5, then $P_N \leftarrow \underline{1} \wedge T = T$ since $f(N',P) \ne \underline{1}$ for any $N' \in \underline{N}$, $P \in \underline{P}$. In this case, $X_N \le P_N = T$ after step A5. Otherwise, suppose this is not the first occurrence of node N in step A5. $X_N \le P_N$ and $X_N \le T \Rightarrow X_N \le P_N \wedge T \Rightarrow X_N \le P_N \leftarrow P_N \wedge T$ after step A5 is executed. Hence, the proposition holds for each execution of step A5. In particular, $X_N \le P_N$ $\forall N \in \underline{N}$ upon termination of the algorithm A. Hence, the theorem is proved since

$$P_N \le X_N \text{ and } X_N \le P_N \Rightarrow X_N = P_N \quad \forall N \in \underline{N}$$ ●