Constant Propagation with Conditional Branches

MARK N. WEGMAN and F. KENNETH ZADECK
IBM T. J. Watson Research Center

Constant propagation is a well-known global flow analysis problem. The goal of constant propagation is to discover values that are constant on all possible executions of a program and to propagate these constant values as far forward through the program as possible. Expressions whose operands are all constants can be evaluated at compile time and the results propagated further. Using the algorithms presented in this paper can produce smaller and faster compiled programs. The same algorithms can be used for other kinds of analyses (e.g., type determination). We present four algorithms in this paper, all *conservative* in the sense that all constants may not be found, but each constant found is constant over all possible executions of the program. These algorithms are among the simplest, fastest, and most powerful global constant propagation algorithms known. We also present a new algorithm that performs a form of interprocedural data flow analysis in which aliasing information is gathered in conjunction with constant propagation. Several variants of this algorithm are considered.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*; *procedures, functions and subroutines*; D.3.4 [**Programming Languages**]: Processors—*code generation*; *compilers*; *optimization*; *preprocessors*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Algorithms, Design, Languages, Theory

Additional Key Words and Phrases: Abstract interpretation, code optimization, constant propagation, control flow graph, interprocedural analysis, procedure integration, static single assignment form, type determination

1. INTRODUCTION

Constant propagation is a well-known global flow analysis problem. The goal of constant propagation is to discover values that are constant on all possible executions of a program and to propagate these constant values as far forward through the program as possible. Expressions whose operands are all constants can be evaluated at compile time and the results propagated further. Using the algorithms presented in this paper can produce smaller and faster compiled programs.

A preliminary version of this paper appeared in Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages, 1985.

Authors' current addresses: Mark N. Wegman, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; F. Kenneth Zadeck, Dept. of Computer Science, Brown University, P.O. Box 1910, Providence, RI 02912.

While the constant propagation problem is easily shown to be undecidable in general (see Kam and Ullman [25], for example), there are many reasonable instances of the problem that are decidable and for which computationally efficient algorithms exist. We present four such algorithms in this paper. Each algorithm presented here is *conservative* in the sense that all constants may not be found, but each constant found is constant over all possible executions of the program.

After some preliminaries, the algorithms are presented in Section 3 in order of increasing power; each successive algorithm finds at least the constants found by the previous algorithm. The first three algorithms are reformulations of the work of others; the fourth is new and contains the best features of each of the previous three. These algorithms are among the simplest, fastest, and most powerful global constant propagation algorithms known. In Section 4 our algorithm is proven to be correct and at least as powerful as the best prior algorithms with polynomial-time bounds. In Section 5 some common implementation problems are discussed.

In Section 6 several techniques are explored to perform constant propagation over an area larger than single procedures. In Section 6.2 the relationship between constant propagation and procedure integration is discussed. Section 6.3 gives a new algorithm that performs a form of interprocedural data flow analysis in which aliasing information is gathered in conjunction with constant propagation.

Section 7 outlines some open problems and Section 8 concludes the paper.

1.1 Uses for Constant Propagation Algorithms

Constant propagation techniques serve several purposes in optimizing compilers:

—Expressions evaluated at compile time need not be evaluated at execution time. If such expressions are inside loops, a single evaluation at compile time can save many evaluations at execution time.

—Code that is never executed can be deleted. Unreachable code (a form of dead code) is discovered by identifying conditional branches that always take one of the possible branch paths.

—Detection of paths never taken simplifies the control flow of a program. The simplified control structure can aid the transformation of the program into a form suitable for vector processing (see Furtney and Pratt [20] and Pratt [31]) or parallel processing (see Ellis [17]).

—Since many of the parameters to procedures are constants, using constant propagation with procedure integration can avoid the expansion of code that often results from naive implementations of procedure integration.

—Constant propagation can be done over a variety of domains, for example, over the type fields of values.

2. PRELIMINARIES

In this section we introduce the mathematical notation used to represent programs and values.

2.1 Graph Definitions

Since an algorithm's performance is usually specified in terms of the size of its input, it is necessary to define some common measures of program size:

N is the number of assignment statements plus the number of expressions whose value is branched on in the program. For notational convenience, each node in the program flow graph contains one expression. This number also closely approximates the number of definition sites in the program since most statements assign a value.

E is the number of edges in the program flow graph. A reasonable approximation for E is twice N, since conditional statements typically have only two successors.[1]

V is the number of variables in the program.

We say that a program consist of three types of nodes: *conditional nodes*, *assignment nodes*, and a unique *start node*.[2] Conditional nodes are potential deviations in control flow through a program. An expression is evaluated at a conditional node and control is subsequently transferred to another node; for simplicity, assume that such expressions have no side effects. Assignment nodes are sites at which variables are defined in terms of other variables and constants; for simplicity, assume that only scalar (i.e., nonsubscripted) variables participate in assignment nodes (see Section 7). In procedures with multiple entry points, the start node has out-edges to any entry node of the procedure.

2.2 The Lattice for Constant Propagation

The output of a constant propagation algorithm is an *output assignment* of lattice values to variables at each node in the program. Let all variables defined or used within a given assignment or conditional node be characterized by a *lattice element* that represents compile-time knowledge about the value of such variables during execution of the algorithm. As depicted in Figure 1, the lattice element can be one of three types: the highest element is *top*, $\top$, the lowest is *bottom*, $\bot$, and all elements in the middle are *constant*, $\mathscr{C}$. There is an infinite number of $\mathscr{C}_i$ lattice elements, each corresponding to a different constant i. In the lattice-theoretic sense, no constant is higher or lower than any other. Each node of the program has cells, called *LatticeCells*, to hold the lattice elements; the values stored in these elements change as the algorithm progresses. The LatticeCells are associated with the results and operands of the expressions; the details of the association depend on the particular algorithm.

[1] This has been experimentally verified by Pratt [31] and Allen [4], and is true because most structured programming constructs give rise to programs with binary branches. One common exception is the computed goto statement in Fortran, which can give rise to graphs with up to N^2 edges if the targets of the goto's are all of the other statements in the program; such programs are very rare.

[2] It is possible that the start node could have a branch to every other statement in the program, giving rise to a program in which the number of edges was three times the number of nodes. Such a program would, however, be extremely rare.

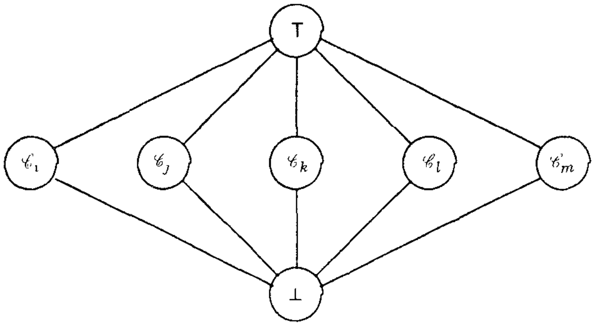


Fig. 1. The three-level lattice.

Fig. 2. Rules for $\sqcap$.

$$\begin{aligned} &\text{any} \sqcap \top = \text{any} \\ &\text{any} \sqcap \bot = \bot \\ &\mathscr{C}_i \sqcap \mathscr{C}_j = \mathscr{C}_i \text{ if } i = j \\ &\mathscr{C}_i \sqcap \mathscr{C}_j = \bot \text{ if } i \neq j \end{aligned}$$

At the end of constant propagation, assigning a constant $\mathscr{C}$ to a LatticeCell means that on all possible executions of the program, the associated result or operand always has the same value when that node is exited. Assigning $\bot$ means that a constant value cannot be guaranteed and assigning $\top$ means that the variable may be some (as yet) undetermined constant. Upon termination of a constant propagation algorithm, all LatticeCells are either $\mathscr{C}$ or $\bot$ at executable nodes.

The algorithms start with the optimistic assignment of $\top$ to the LatticeCell of all operands of expressions at all nodes except the start node. If the variable has an explicit initializer or if the language specifies an implicit initializer (e.g., in LISP, all cells are initialized to `nil`), then that value is used at the start node. The algorithms may obtain better information if $\top$, rather than $\bot$, is assigned to the values of uninitialized variables at the entry of the program. In languages in which the use of an uninitialized variable is allowed but undefined (as in FORTRAN), the use of $\top$ may make the analysis yield an incorrect result. Thus, each uninitialized variable must be assigned $\bot$. On the other hand, if the language forbids such uses, the uninitialized variables may be assigned $\top$.

The algorithms proceed by lowering (in the lattice-theoretic sense) the LatticeCells of the operands and results at each node as more information is discovered, a process that continues until a fixed point is achieved. The additional information is inserted by applying the meet ($\sqcap$) rules shown in Figure 2, where each of the operand values at a node corresponds to the conditions prior to the execution of the statement. These rules ensure that the value at the join point is no higher than the value entering from any of the predecessors.

```
any ∨ true  = true
any ∧ false = false
```

Fig. 3. Special rules for ∧ and ∨.

```
j ← 5
if i = j then i ← i + 1
```

Fig. 4. A conditional branch where information about i can be derived.

If a variable is not the target of an assignment statement in a node, its value is unchanged by the node. If the node is an assignment statement, the value of the result and the value of operands of other nodes may change and are reevaluated according to the *expression evaluation rules*: the value of the operands of an expression corresponds to the value of the variables at the entrance to the node, and the result corresponds to the value of variables that change during the execution of the node.

Usually, if the node is an assignment and any of the variables used in its expression portion has a value of $\perp$, the value exiting the assignment statement for that variable is $\perp$. If all values used in its expression portion are constant, the value of the assigned variable is the value of the expression when evaluated with those constant values. Otherwise the value assigned is $\perp$.

For certain operators, however, we can give special expression rules that yield better information. For example, if the operator is an ∨ and one of the operands is known to be true, then the value of the expression is true whether or not the other operand is $\perp$. These rules are given in Figure 3.

Information can sometimes be derived from the equality tests that control conditional branches [2]. On entrance to the then branch in Figure 4, the value of i is 5. We can modify the program so that we can derive this information by inserting extra nodes containing assignments between the conditional and the entrance to the then branch. The variable i is assigned the join of the LatticeCells of i and j (a similar assignment to j is also added). These are not assignment statements in the ordinary sense, because they are not used to cause changes in the program's state and need not be executed; rather, they are used to model changes in the assertions about the program state and, thus, are in the intermediate code.

In the algorithms below, the changes in the LatticeCells associated with the operands may require an expression to be reevaluated many times. In Section 5.4, we show how to do all of the reevaluations of a given expression in time proportional to the size of the expression.

3. CONSTANT PROPAGATION ALGORITHMS

In this section we present four algorithms for determining constants. They are described in order of increasing power; each algorithm finds at least the constants found by the previous algorithm. These algorithms are among the simplest, fastest, and most powerful global constant propagation algorithms known. The first three algorithms are reformulations of the work of others;

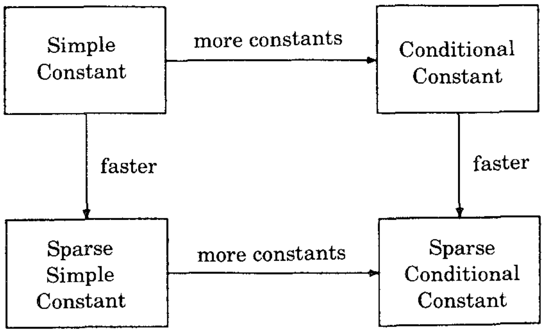


Fig 5. Relationship among the four constant propagation algorithms.

the fourth is new and contains the best features of each of the previous three. Figure 5 shows the relationship among the four algorithms.

The first algorithm, *Simple Constant* (SC), was developed by Kildall [26] and is presented in Section 3.1. Kildall was among the first to describe the constant propagation problem and to give an algorithmic solution.

The second algorithm, *Sparse Simple Constant* (SSC), is an easily understood reformulation of an algorithm developed by Reif and Lewis [32] and is presented in Section 3.2. This algorithm uses a recently developed data structure called the *static single assignment* graph (SSA graph) [16]. The SSA graph is a variant of the global value graph of Reif and Lewis [32], which in turn is based on the *p-graph* of Shapiro and Saint [40]. The SSA graph allows this algorithm to find a class of constants equivalent to those of SC, yet the algorithm is faster than SC by a factor proportional to the number of variables in the program. Indeed, the speedup can be proportional to the product of the number of variables in the program and the number of edges in the program flow graph. It is unfortunate that this algorithm was not recognized for many years, since it works in time linear in the size of the SSA graph.

The third algorithm, *Conditional Constant* (CC), is a variant of Wegbreit's Algorithm 3.1 [42] and is presented in Section 3.3. CC discovers all constants that can be found by evaluating all conditional branches with all constant operands, but it uses the same input data structures and is asymptotically as slow as SC. The attraction of CC is that it propagates the values in such a way that when conditional branches are found to have a constant conditional expression, the search for constants can ignore parts of the program that are never executed. The algorithm does unreachable code elimination in combination with constant propagation. The first benefit of this approach is that the algorithm may run faster than SC, since it need not evaluate the sections of the program that are never executed. A second benefit is that values created in the unreachable areas cannot possibly kill potential constants, and thus CC can find more constants than can SC.

The fourth algorithm, *Sparse Conditional Constant* (SCC), is new and is presented in Section 3.4. SCC finds the same class of constants as CC, yet has the same speedup over CC as SSC has over SC.

3.1 Simple Constant

Kildall's *Simple Constant* (SC) algorithm [26] uses the program flow graph for propagation of values. At each node in the program, two LatticeCells are associated with the value of every variable in the program, one with the value at entry to the node and the other with the exit. The process of visiting a node involves examination of every LatticeCell at that node. Initially the start node is placed on the worklist. A node is chosen from the worklist, removed from the worklist, and examined. The lattice value stored at the entry to the examined node becomes the meet of values at the exits of preceding nodes. The statement is evaluated on the basis of the new entering values. This may cause the value of a variable that it is assigned to in the node to differ from the value associated with the variable at the exit LatticeCell. In that case all nodes following the examined node must be examined, and they are added to the worklist. If no exit LatticeCells change, then no nodes are added to the worklist. The process repeats until the worklist is empty.

SC finds those constants that Kildall calls *simple constants*. Simple constants are all values that can be proved to be constant subject to two constraints: no information is assumed about which direction branches will take, and only one value for each variable is maintained along each path in the program.

Since the lattice value of each variable can only be lowered twice, each node may be visited at most $2 \times V \times I$ times, where I is the number of in-edges into that node. Thus, the time required for Kildall's algorithm is $O(E \times V)$ node visits and V operations during each node visit. This results in a worst-case running time of $O(E \times V^2)$. The best-case running time may be $O(E \times V)$. It is our intuition that the worst case is rarely achieved. The space required is $O(N \times V)$.

We have described the problem in somewhat different terms from Kildall. In Kildall's lattice, each lattice element corresponds to the state of all variables in the program. In our lattice, each state corresponds to the state of a single variable. By using our representation and a technique described in Section 5.4, it is possible to lower the time bound to $O(E \times V)$ changes to variable values, which is the best one can hope to get out of an approach like this. These optimizations are not obvious in the context presented by Kildall.

We have initialized the worklist with only the start node, where Kildall started with all of the nodes on the worklist. This is consistent with and required for the other algorithms presented in this paper, but does not affect SC in any significant way.

When SC visits a node, it applies a function that takes as input the value of all variables at the entrance of the node and produces the set of values for all variables at the exit of the node. In the next section we reformulate that notion in order to model more closely what Reif and Lewis [32, 33] did and what we do. This reformulation of SC allows the SSC algorithm to run faster than SC, but produces the same information as SC.

3.2 Sparse Simple Constant

Reif's and Lewis's *Sparse Simple Constant* (SSC) algorithm [32, 33] finds all simple constants. It achieves a speedup over SC by using a sparse representation (in the presentation here, the SSA graph) to propagate the values through the program. The class of constants found is unchanged.

3.2.1 *The Static Single Assignment Graph.* In SSA form, the program is transformed so that only one assignment can reach each use. For straight-line programs, the transformation to SSA form is straightforward. Each assignment to a variable is given a unique name (shown as a subscript in Figure 6) and all of the uses reached by that assignment are renamed to match the assignment's new name. More complicated programs have branch and join nodes. At the join nodes, we must add a special form of assignment called a *ϕ-function*. A ϕ-function at the entrance to a node X has the form $V \leftarrow \phi(R, S, \ldots)$, where $V, R, S, \ldots$ are variables. The number of operands $R, S, \ldots$ is the number of control flow predecessors of X. The predecessors of X are listed in some arbitrary fixed order, and the jth operand of ϕ is associated with the jth predecessor. If control reaches X from its jth predecessor, then V is assigned the value of the jth operand. Each execution of a ϕ-function uses only one of the operands, but which one depends on the flow of control just before entering X. Any ϕ-functions at X are executed before the ordinary statements in the node that contributes X to the program flow graph. Figure 7 shows the use of ϕ-functions in SSA form and Figure 8 gives the SSA form of a simple program with loops.

In general, two separate steps are required to translate a program into SSA form. In the first step, some trivial ϕ-functions $V \leftarrow \phi(V, V, \ldots)$ are inserted at some of the join nodes in the program flow graph. In the second step, new variables V_i (for $i = 0, 1, 2, \ldots$) are generated to serve as new names for each variable V. Each mention of V in the program is replaced by a mention of one of the new names V_i. (A *mention* may be on either side of an assignment statement and may be in an ordinary assignment or in a ϕ-function.) A program is defined to be *in SSA form* if, for every original variable V, trivial ϕ-functions for V have been inserted and each mention of V has been changed to a mention of a new name V_i such that the following conditions hold:

(1) If a program flow graph node Z is the first node common to two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ that start at nodes X and Y containing assignments to V, then a ϕ-function for V has been inserted at Z.

(2) Each new name V_i for V is the target of exactly one assignment statement in the program text.

(3) Along any program flow path, consider any use of a new name V_i for V (in the transformed program) and the corresponding use of V (in the original program). Then V and V_i have the same value.

```
v ← 1              $v_1$ ← 1
... ← v + 1        ... ← $v_1$ + 1
v ← 2              $v_2$ ← 2
... ← v + 2        ... ← $v_2$ + 2
```

Fig. 6. Straight-line code and its single assignment version.

Fig. 7. An if - then - else and its single assignment version.

```
if P                 if P
  then v ← 1           then $v_1$ ← 1
  else v ← 2           else $v_2$ ← 2
                     $v_3$ ← $\phi$($v_1$, $v_2$)
... ← v + 2          ... ← $v_3$ + 2
```

```
i ← 1                      $i_1$ ← 1
j ← 1                      $j_1$ ← 1
k ← 1                      $k_1$ ← 1
while (P)                  while (P)
                             $j_2$ ← $\phi$($j_4$, $j_1$)
                             $k_2$ ← $\phi$($k_5$, $k_1$)
  if (Q)                     if (Q)
    then do                    then do
      j ← i                      $j_3$ ← $i_1$
      k ← k + 1                  $k_3$ ← $k_2$ + 1
    end                        end
    else k ← k + 2             else $k_4$ ← $k_2$ + 2
                             $j_4$ ← $\phi$($j_3$, $j_2$)
                             $k_5$ ← $\phi$($k_3$, $k_4$)
end                        end
```

Fig. 8. A simple program and its single assignment version.

Once a program is in SSA form, we add connections called *SSA edges*. Each connection goes from the unique point where a variable is given a value to a use of that variable. SSA edges are essentially def-use chains [1] in the SSA program.

A program is in *minimal SSA form* if it is in SSA form and if the number of ϕ-functions inserted is as small as possible, subject to Condition 1 above. The optimizations that depend on SSA form are still valid if there are some extraneous ϕ-functions beyond those that would appear in minimal SSA form. However, extraneous ϕ-functions sometimes inhibit other optimizations by concealing useful facts [5], and they always add unnecessary overhead to the optimization process itself. Thus it is important to place ϕ-functions only where they are required.

For any variable V, the program flow graph nodes at which we should insert ϕ-functions in the original program can be defined recursively by Condition 1 in the definition of SSA form. A node Z *needs a ϕ-function for V* if Z is the first node that two nonnull program flow paths have in common, when those two paths originate at two different nodes containing assignments to V or needing ϕ-functions for V. Nonrecursively, we may observe that a node Z needs a ϕ-function for V because Z is the first node common to two nonnull paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ that start at nodes X and Y containing assignments to V. If Z does not already contain an assignment to V, then the ϕ-function inserted at Z adds Z to the set of nodes that contain assignments to V. With more nodes to consider as origins of paths, we may observe that more nodes appear as the first node common to two nonnull paths originating at nodes with assignments to V. The set of nodes observed to need ϕ-functions thus gradually increases until it stabilizes. When ϕ-functions are placed in this way, minimal SSA form can be obtained by an easy adaptation of well-known def-use chaining. The algorithm presented in [16] obtains the same end results as this brute-force approach, but it places the ϕ-functions and performs the renaming in much less time than brute force would require.

Minimal SSA form is a refinement of Shapiro's and Saint's [40] notion of a pseudo-assignment. The *pseudo-assignment nodes* for V are exactly the nodes that need ϕ-functions for V. For a program flow graph with E edges describing a program with V variables, one algorithm [34] requires $O(E\alpha(E))$ bit vector operations (where each vector is of length V) to find all the pseudo-assignments. A simpler algorithm [38] for reducible programs computes SSA form in time $O(E \times V)$. Both of these algorithms are effectively quadratic, and the algorithm in [38] sometimes uses extraneous ϕ-functions. The method presented in [16] is effectively linear in the size of the program although there are cases where it behaves nonlinearly.

3.2.2 *The Algorithm.* SSC works as follows:[3]

(1) Examine all expressions. If it is not possible that the value of an expression will be evaluated at compile time (for example, a read statement), then the corresponding LatticeCell is assigned $\bot$. If no variables appear in the expression, then the expression is evaluated and the LatticeCell is assigned an appropriate $\mathscr{C}_i$. All other LatticeCells are assigned $\top$.

A worklist is initialized to contain all SSA edges where the definition is from an expression that has a LatticeCell that is not $\top$.

(2) The algorithm terminates when the worklist becomes empty.

(3) An SSA edge is taken off the worklist. We form the meet of the value of the LatticeCell at the definition end of the SSA edge and the value of the LatticeCell at the use end of the SSA edge. The meet is performed under the rules given in Section 2.2.

(4) If the meet of the values is different from the value at the use end, then the use end is replaced by the meet. The new value is used to recompute the value of the expression in which the previous value had been used, again according to the expression rules in Section 2.2. If the new value for the expression is lower than the value stored for the expression, then all SSA edges with their source at this node are added to the worklist.

[3] SSC as originally presented used a global value graph as its sparse representation; here we use the SSA graph, which is a variant. These two representations produce equivalent results when used with this algorithm.

3.2.3 *Asymptotic Complexity.* The time complexity of this algorithm is proportional to the size of the SSA graph. The SSC algorithm requires that each SSA edge be examined at least once and at most twice. The examinations occur when the value of its definition site is lowered to either $\mathscr{C}$ or $\bot$. In theory the size can be $O(E \times V)$, but empirical evidence indicates that the work required to compute the SSA graph is linear in the program size [16]. Thus, we expect our algorithm to be linear in practice.

3.3 Conditional Constant

Wegbreit's Algorithm 3.1 [42] is a general algorithm for performing global flow analysis that takes conditional branches into account. In this section, we specialize Wegbreit's general algorithm to perform constant propagation and call the result *Conditional Constant* (CC). CC is more powerful than SC because, whenever CC can assume that a conditional expression is always constant, it assumes that the branch it guards goes in only one direction.

Consider the example in Figure 9. Neither SC nor SSC is capable of discovering that j = 1 since they make no assumptions about the possible branch directions. Since *i* is always 1, however, the condition always takes the true branch and j is always equal to 1. Such code may be the result of procedure integration or abstract data type compilation.

To exploit this knowledge about conditional branches, we do not propagate values along all program flow graph edges, as in SC. Rather, CC defers the evaluation of any program flow graph edge until it is marked as executable. Each program flow graph edge is initially marked as not executable. Program flow graph edges are marked executable by symbolically executing the program, beginning with the start node. Whenever an assignment node is executed, the out-edge in the program flow graph leaving that node is marked as executable and added to the worklist. Whenever a conditional node is executed, the expression controlling the conditional is evaluated and we determine which branch(es) may be taken. If the expression evaluates to $\bot$, then all branches may be taken. The edges corresponding to these branches are added to the worklist. If the expression evaluates to $\mathscr{C}$, only one branch can be taken, and the associated edge is added to the worklist.

This algorithm is able to ignore any definition that reaches a use via a program flow graph edge that is never executed. Thus, this algorithm accomplishes a form of *dead code elimination* called *unreachable code elimination*.[4]

This algorithm has the same asymptotic running time as SC, $O(E \times V^2)$ in the worst case in which no branches are found to be constant. This algorithm is expected to have better average-case complexity, however, since it can ignore parts of the program that will never be executed and since SC should behave better than $O(E \times V^2)$.

[4] Two classical techniques are called dead code elimination. The goal of the first, *unreachable code elimination*, is to eliminate code that can never be executed. The goal of the other, *unused code elimination*, is to delete sections of code whose results are never used (see Allen and Cocke [2]). Each of these techniques finds a different class of dead code, and neither subsumes the other.

```
i ← 1
...
if i = 1
  then j ← 1
  else j ← 2
```

Fig. 9. A conditional constant definition.

Many optimizing compilers repeatedly execute constant propagation and unreachable code elimination since each provides information that improves the other. CC solves this problem in an elegant way by combining the two optimizations. Additionally, the algorithm gets better results than are possible by repeated applications of the separate algorithms, as described in Section 5.1.

3.4 Sparse Conditional Constant

We wish to derive a version of CC that also improves running time, just as SSC was derived from SC to improve running time. In order to do this, we must utilize some of the special properties of the SSA graph. We call this algorithm *Sparse Conditional Constant* or SCC.

When the SSA graph was constructed, ϕ-functions were inserted at some join nodes. The meaning of a ϕ-function is that if control reaches the node in the program flow graph along its ith in-edge, the result of the ϕ-function is the value of its ith operand.

In the SSC algorithm, when the meet rule was applied to a ϕ-function, the meet operator was applied to all of the operands of the ϕ-function. In the SCC algorithm, the meet operator is applied only to those operands of the ϕ-function that correspond to the program flow graph edges marked executable. Those that are not executable effectively have the value of $\top$.

This algorithm uses two worklists: *FlowWorkList* is a worklist of program flow graph edges and *SSAWorkList* is a worklist of SSA edges.

SCC works as follows:

(1) Initialize the FlowWorkList to contain the edges exiting the start node of the program. The SSAWorkList is initially empty.

Each program flow graph edge has an associated flag, the *ExecutableFlag*, that controls the evaluation of ϕ-functions in the destination node of that edge. This flag is initially `false` for all edges.

Each LatticeCell is initially $\top$.

(2) Halt execution when both worklists become empty. Execution may proceed by processing items from either worklist.

(3) If the item is a program flow graph edge from the FlowWorkList, then examine the ExecutableFlag of that edge. If the ExecutableFlag is `true` do nothing; otherwise:

  (a) Mark the ExecutableFlag of the edge as `true`.

  (b) Perform Visit-ϕ for all of the ϕ-functions at the destination node.

  (c) If only one of the ExecutableFlags associated with the incoming program flow graph edges is `true` (i.e., if this is the first time this node has been evaluated), then perform VisitExpression for the expression in this node.

(d) If the node only contains one outgoing flow graph edge, add that edge to the FlowWorkList.

(4) If the item is an SSA edge from the SSAWorkList and the destination of that edge is a ϕ-function, perform Visit-ϕ.

(5) If the item is an SSA edge from the SSAWorkList and the destination of that edge is an expression, then examine ExecutableFlags for the program flow edges reaching that node. If any of them are true, perform VisitExpression. Otherwise do nothing.

The value of the LatticeCell associated with the output of a ϕ-function is defined to be the meet of all arguments whose corresponding in-edge has been marked executable. It is computed by Visit-ϕ. Visit-ϕ is called whenever the value of the LatticeCell associated with one of its operands is lowered or when the ExecutableFlag associated with one of the in-edges becomes true.

Visit-ϕ is defined as follows: The LatticeCells for each operand of the ϕ-function are defined on the basis of the ExecutableFlag for the corresponding program flow edge.

executable The LatticeCell has the same value as the LatticeCell at the definition end of the SSA edge.

not - executable The LatticeCell has the value $\top$.

VisitExpression is defined as follows: Evaluate the expression obtaining the values of the operands from the LatticeCells where they are defined and using the expression rules defined in Section 2.2. If this changes the value of the LatticeCell of the output of the expression, do the following:

(1) If the expression is part of an assignment node, add to the SSAWorkList all SSA edges starting at the definition for that node.

(2) If the expression controls a conditional branch, some outgoing flow graph edges must be added to the FlowWorkList. If the LatticeCell has value $\bot$, all exit edges must be added to the FlowWorkList. If the value is $\mathscr{C}$, only the flow graph edge executed as the result of the branch is added to the FlowWorkList.[5]

3.4.1 *Asymptotic Complexity.* As in SSC, each SSA edge can only be examined twice. Nodes in the program flow graph are visited once for each of their in-edges. The asymptotic running complexity of this algorithm is the number of edges in the flow graph plus the number of SSA edges and should be linear in practice.

CC may be impractically slow and, consequently, was ignored for a long time. Many workers in code optimization had tried to derive practical sparse algorithms that achieved CC's results. However, they started from the sparse

[5] The value cannot be $\top$, since the earlier step in VisitExpression will have lowered the value.

representation then prevailing, def-use chains without SSA form. Def-use chains without SSA form do not determine the best information, as described in Section 5.2.

4. THEOREMS AND PROPOSITIONS

In this section we first show that SCC is conservative, that is, it does not label as constant variables that are not constant. We then show that SCC is at least as powerful as CC and finds all the constants that CC does. Wegbreit's Algorithm 3.2 and Holley's and Rosen's algorithm [23] are more powerful than CC but may require exponential time.

Before we define "conservative" more formally, we need the concept of an executable sequence.

Definition. An *executable sequence* is a sequence of tuples in which each tuple consists of a node in the flow graph and a lattice element for each variable. The first tuple contains the start node, and each subsequent tuple is derived by evaluating the expression at the node and changing values as appropriate. The next tuple is determined by deriving values of expressions and branching appropriately.

Definition. An output assignment is *conservative* if the values of variables at each node are no higher in the lattice-theoretic sense than the values when that node is reached on any executable sequence.[6]

THEOREM. *The output assignment derived from* SCC *is conservative and an execution sequence traverses only those edges* SCC *labels as executable.*

PROOF. Suppose the contrary. Then there must be a shortest execution sequence that either traverses an edge not labeled executable or has a value lower than the output assignment to a variable at that node. If an edge not labeled executable is on the path, it must be the first such edge or else there is a shorter such path. The preceding node on the path must be a conditional and the values of the variables used in the condition must be different in the sequence from those in the output assignment, since otherwise the edge would have been labeled executable. But then there is a shorter path.

Thus, there must be a value which is higher in the output assignment than on the sequence. That value is used at the last node in the shortest sequence. Therefore there must be a preceding assignment node to the variable, and at the assignment node the values agree with the output of SCC. There must be an SSA edge from that assignment node to the last node in the sequence, since there are no intervening assignment nodes. Thus the value at the last node must be correct. □

[6] This notion is very similar to Graham and Wegman's *safe assignment* [21]. In [21], an executable sequence can be any path, even one that takes branches contrary to provably constant branches. Thus, a safe assignment is conservative, but a conservative assignment is not necessarily safe.

Now that we have shown that no more constants are found than is correct, we wish to show that we find as many constants as other algorithms. We compare our algorithm to CC, which finds more constants than SC. CC is identical to SC except that CC does not propagate values along branches from conditions until it shows that the branch may be taken.

THEOREM. *The output assignment derived by SCC gives each variable at each node a value which in the lattice-theoretic sense is at least as large as CC's output assignment.*

PROOF. Suppose the contrary. There must be a lower value at an executable node, since if the two algorithms agree on which way branches can go, they must agree on which nodes are executable.

Then SCC has a shortest execution in which it evaluates an expression whose value is lower than CC's value or a point at which a variable is assigned a lower value. If the assignment is a normal assignment (does not use a ϕ-function), then one of the variables involved in the expression must have a lower value; hence there is an earlier assignment. If the assignment is from a ϕ-function, then either (1) one of its arguments is lower than it would be in the preceding node in CC or (2) the argument is flagged as executable but the entering edge is not executable in CC.

In case (1), there must be an earlier assignment to the variable. In case (2), there must be an edge that we say is executable but CC does not. If so, we must have evaluated the expression controlling the condition to a lower value than CC. But that evaluation takes place earlier, and hence there is a shorter path. □

5. OBSERVATIONS ON THE CONSTANT PROPAGATION ALGORITHMS

Several problems can arise in implementations of constant propagation algorithms that affect the quality of the solution (the number of constants found) as well as the asymptotic complexity.

5.1 Finding the Maximal Fixed Point

SSC was not the first algorithm to use a sparse representation for propagation; it was, however, the first to use such a representation and to find all simple constants. Many global constant propagation algorithms in common use resemble SSC but do not achieve the same results. In these weaker algorithms, propagation at a use is deferred until all edges that reach that use have been visited. If they all have the same value, the join node is given that value. These algorithms start with the assumption that all expressions have $\perp$ and attempt to raise the lattice value to $\mathscr{C}$ when it can be proven that all values reaching that location are constant. Each SSA edge is visited only once. The weaker algorithms are *pessimistic* in their propagation: They never propagate any value unless they are certain that the value will never be invalidated.

Fig. 10. A simple program loop.

```
ı ← 1
while (...) do
  ȷ ← i
  i ← f(...)
  ...{no stores are done into ȷ here}
  i ← ȷ
end
```

All of the algorithms presented in this paper, on the other hand, are *optimistic*. They start with the possibly incorrect assumption that everything may be constant and determine the values that may not be constant. If an optimistic algorithm is stopped before it terminates naturally, the information gathered may be wrong. Pessimistic algorithms may be stopped at any time and still produce correct (though poor) results.

The major drawback of the pessimistic approach is that propagation cannot proceed around cycles in the SSA graph. These cycles are typically the result of simple loops in the program. SSC finds more constants than the weaker algorithms, since SSC can propagate through loops. In Figure 10, the variable i always has the value 1 at the bottom of the loop. The weaker algorithms get stuck on the loop and fail to discover this constant.

In practice, the difference between the optimistic and pessimistic versions of SC or SSC would be quite small. When conditional branches are considered, however, as in CC or SCC, the differences could be significant, particularly when these algorithms are used for procedure integration or type determination, as discussed in Section 6.2.

More precisely, these weaker algorithms find a *fixed point* that is not *maximal*. A fixed point is defined by saying that:

(1) The lattice element at each node represents what is provably true at a node.

(2) We have functions whose domain and range is the lattice and these functions represent what changes in state take place when we go from node to node.

(3) If f is a function associated with the transition from node u to node v, then the lattice element at v is less than or equal to (in the lattice-theoretic sense) f applied to the lattice element at u.

The maximal fixed point is the fixed point with the largest lattice elements at the nodes and has been used as a minimal acceptable criterion of the quality of flow analysis (see Kam and Ullman [25] and Graham and Wegman [21]). Several pessimistic versions of SSC have been published and implemented. These include an algorithm described by Kennedy in chapter 6 of [14], an algorithm in chapter 4 of [30], and an algorithm by Kennedy in chapter 1 of [28]. Two optimistic versions of SSC have been published, the first by Reif and Lewis [32, 33] and the second by Ferrante and Ottenstein [19].

Several versions of SC have also been published. The first, by Kildall [26], also appears in [1]; a generalization of this was published by Kam and

```
(a) ı ← 1
    j ← 2
if  j = 2
(b)   then i ← 3
(c) ... ← i
```

Fig. 11. Constant not found with def-use chains.

Ullman [25]. Each of these is optimistic, although none has made the observations made here to reduce the worst case complexity from cubic to quadratic.

5.2 Def-Use Chains

Many constant propagation algorithms work on a graph of *def-use chains*. This data structure is common in optimizing compilers and is described in many textbooks on compilers, such as the one by Aho, Sethi, and Ullman [1]. Def-use chains can cause two problems with the algorithms presented here.

A def-use chain is a connection from a *definition site* for a variable to a *use site* for that variable. A definition site for a variable is a statement that assigns to the variable. A use site is normally an operand of an expression. There is a def-use chain between a definition site and a use site if the use site can be reached from a definition site along the program flow graph without passing through another definition site for that variable.

If SCC is performed using def-use chains rather than the SSA graph, some constants are missed, because def-use chains exist along paths that are not executable. Consider the program shown in Figure 11. Statement (b) provides the only possible value for statement (c), because the path through (b) is the only path to (c) (we know this because the condition must always be true). A def-use chain version of SCC does not find this because it applies the def-use chain that starts from (a) and the algorithm does not know that the value is really killed by (b).

Another shortcoming of def-use chains is related to the size of the graph. In def-use chains, many definitions can reach a use. The number of def-use chains for a single variable can be N^2, and thus the worst case complexity of an algorithm that uses def-use chains is $N^2 \times V$. In the SSA graph, however, only one definition reaches each use, and only N ϕ-functions can be inserted for each variable. This means that the worst case complexity of a constant propagation algorithm that uses the SSA graph is only $N \times V$.

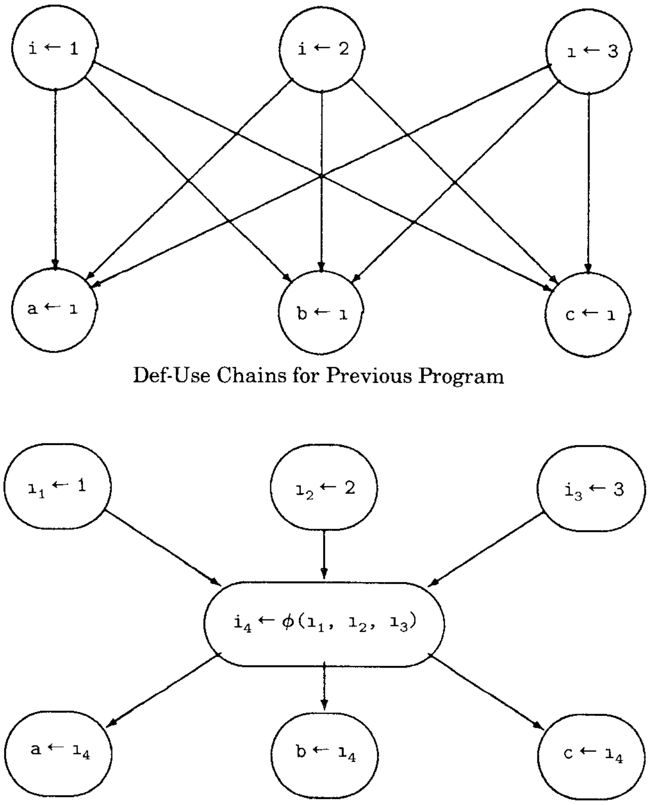
Figure 12 shows a program in which the def-use chain graph grows as N^2. Here each of several definition sites for each variable reaches each use site for each variable. This does not occur in the SSA graph, since a ϕ-function is inserted at the join node.

5.3 Nodes versus Edges

In the algorithms presented here, the ExecutableFlag is associated with the program flow graph edges rather than the nodes. Two nodes may be executable and there may be an edge between them, but that edge may not be traversable. In Figure 13, if `p` can be determined always to be `false`, then `i`

```
select j
  when x {i ← 1}
  when y {i ← 2}
  when z {i ← 3}
end
select k
  when x {a ← i}
  when y {b ← i}
  when z {c ← i}
end
```

Original Program



Def-Use Chains for Previous Program

SSA Graph for Previous Program

Fig. 12. Worst-case behavior of def-use chains.

will be 10 after execution of the loop. If the ExecutableFlag is associated with the nodes rather than the edges in the graph, then i will have the value ⊥ at the end of the loop.

An alternative way of implementing this would be to add nodes to the graph and then associate an ExecutableFlag with each node. An additional node must be inserted between any node that has more than one immediate

```
i ← 1
while (true) do
  if p
    then exit
  ı ← i + 1
  if i = 10
    then exit
end
print i
```

Fig. 13. Case missed if information is associated with nodes.

successor and any successor node that has more than one immediate predecessor. Such a transformation has the effect of changing `if - then` statements to `if - then - else` statements and also adds nodes at the destinations of gotos and loop–exits. Since this transformation also improves redundancy elimination algorithms [27, 13, 38], it may be the method of choice.

5.4 Expression Evaluation

An expression may be evaluated twice for each of its operands, since the LatticeCell associated with each operand may be lowered twice. If the expression is large, this can be expensive. Reif and Lewis [32] store expressions as trees and evaluate the leaves and internal nodes of the tree only as their values change, and we suggest doing the same. As most expressions in most programs are small, however, this improvement may be of only theoretical interest.

6. PROCEDURE INTEGRATION AND INTERPROCEDURAL CONSTANT PROPAGATION

Code optimization techniques are useful because it is desirable to program at a high level of abstraction and use automatic techniques to specialize the generic routines implementing the abstractions. Generic routines are usually implemented by procedures. The techniques discussed here are also applicable to generic methods in object-oriented programs.

Two techniques can be used to specialize generic procedures; the first is based on procedure integration and the second on interprocedural analysis. The distinctions between procedure integration and interprocedural analysis are important because

(1) at any particular call site, the full power of a generic routine may be unnecessary. Procedure integration provides separate copies of the procedure, *each* of which may be specialized differently for their respective call sites. Interprocedural analysis allows specialization of the procedure only in ways that are simultaneously appropriate to *all* sites.

(2) procedure integration breaks down the barrier between the call site and the called procedure. More powerful specialization techniques are available after procedure integration than after interprocedural analysis. For example, if the call site is inside a loop, a code motion algorithm may be able to move some of the code in the generic routine to a point outside the loop.

```
(setq i 1)
loop
  (cond ((greaterp i 10) (go out)))
  ...
  (setq i (plus i 1))
  (go loop)
out

; plus is a macro which expands in line to the following. It replaces
; x with the first argument to the macro and y with the second.

  (cond
    ((and (integer x) (integer y))
      (integeradd x y))
        ;The result of integeradd is an integer.
    ...
  )
```

Fig. 14. Removal of conditional type check.

(3) procedure integration may require an unacceptably large amount of space. In the case of recursion, a naive procedure integration algorithm may require an infinite amount of space.

To gain some intuition about kinds of code that might be integrated and the kinds of problems that might be encountered, consider the code fragment in Figure 14. Here the plus routine is a macro expanded by the compiler. We view the type field of a variable as a variable in its own right and perform constant propagation on the type field. This is a form of type inference. At compile time, the conditional expression for the execution-time type check can be eliminated. Our analysis proves that I is always an integer. However, to determine this, the branches of code producing floating point results must be eliminated. This can only be done when analysis shows that only integer values are passed in from the call site. Of course the analysis must optimistically assume that the arguments are integers.

In Section 6.1 we describe how to represent the semantics of procedure calls using SSA form. In Section 6.2 we show how constant propagation can be profitably combined with procedure integration. In Section 6.3 we give an algorithm to perform interprocedural constant propagation without encountering the space problems arising from procedure integration.

6.1 Procedure Calls in SSA Form

Procedure calls have two effects. One is to transfer control to a procedure and the subsequent return; the other is to create instances of and to change the names of variables. SSA form easily models the control flow aspects of a procedure call, but changes may be required to model the passing of parameters to procedures.

The value returned from a function call is used as a normal expression value in the called procedure.

Each call-by-value parameter is modeled in SSA form by a single assignment statement. The actual parameters are simply assigned to the formal

```
a ← 1           a ← 1
                if IsAliased(a,b)
                  then b ← a
b ← 2           b ← 2
                if IsAliased(a,b)
                  then a ← b
?← a            ?← a
```

Fig. 15. Effect of potential alias (a, b) and insertion of resulting conditional assignments.

parameters, which are new variables in the called procedure. No variables at the call site can be affected.

Each call-by-value-return parameter is modeled in SSA form by two assignment statements. The first assigns the actual parameter to the formal parameter, and the second assigns the formal parameter back to the actual parameter; because we are using SSA form, this second assignment is implemented as an assignment to a new variable.

Each call-by-reference parameter is modeled as a call-by-value-return parameter, with one change: We must deal with problems caused by potential aliasing. Aliasing occurs when it is possible to generate two or more names for the same memory location. In many programming languages, aliasing limits where and to what extent certain optimizations can be performed. Storing into memory by way of one name may affect the value referenced by the other name. (The term "aliasing" can also mean the use of two expressions that, because of pointers, can refer to the same location. This problem is not addressed here.) The remainder of this section describes how to deal with aliasing.

Several program constructs involving procedure calls may give rise to potential aliases:

(1) A variable may be passed by reference in two or more parameter positions of a single call to a subroutine.

(2) A variable that is global to a procedure may be used by its name and also by a reference parameter in the call to the procedure.

(3) Any variable that is aliased to an argument to a procedure is also aliased to the parameter.

Aliasing information can be represented as a list of pairs of variable names for each procedure. If a pair (a, b) is a *must*-alias, then on *all* possible executions of the program, a and b *must* refer to the same storage location. Must-aliasing of (a, b) implies that any assignment to either a or b is an assignment to both a and b. If a pair (a, b) is a *may*-alias, then on *some* execution of the program, a and b *may* refer to the same storage location. Aliasing is not necessarily transitive: The existence of may-alias pairs (a, b) and (b, c) does not imply the existence of (a, c). May-aliasing of (a, b) means that when a store to b occurs along a path from a store into A to a use of a, the value of b may reach the use of a (see the left side of Figure 15). On some executions, a may have the value 1 and on others it may have the value 2. (The detection of potential aliases for a single procedure depends on the semantics of the particular language, and is beyond the scope of this paper.

For example, in some languages two variables cannot be aliased if they are of different types.)

Without some sort of interprocedural analysis, the number of potential aliases in a program is quite large, even though the number of actual aliases is small in practice. Banning [8] gives a simple procedure for computing a conservative estimate of the potential aliases. A variable can be viewed as aliased to itself. A formal parameter that is passed using call-by-reference can be aliased to any variable to which the actual parameter is aliased. One need only keep track of variables that are visible. Banning has created a worklist algorithm that is linear in the number of aliased pairs. Since aliasing is rare in real programs, this algorithm can be assumed to be efficient. Other algorithms exist; see Barth [9], Rosen [37], Myers [29], Cooper [15], and Burke [10].

Aliasing information may be represented in SSA form by inserting if - then structures after each assignment to a variable that is either may- or must-aliased, as shown on the right side of Figure 15. The statement on the then side is an assignment statement from the variable just defined to the variable to which it is aliased. This has the effect of joining the two aliased variables with a ϕ-function. Once these additional if - then structures have been inserted, the SSA graph can be built as before. In Figure 15, two SSA edges would exist; each of these edges would go to a ϕ-function that merges the potentially aliased variables.

The variable IsAliased(a,b) is evaluated as true if the pair is must-aliased. The assignment blocks any values of the aliased variable from reaching beyond the assignment of the first variable. The variable IsAliased(a,b) is evaluated as $\perp$ if the pair is may-aliased. Both values for the second variable are merged in the ϕ-function that is inserted after the if - then. Such a definition is conservative; it is correct under all possible executions, whether or not the alias actually occurs.

The advantage of representing aliasing by if - then structures is that, as other types of analysis (such as constant propagation combined with procedure integration) proceed, the value of the IsAliased variables can often be determined to be true or false. Hence, if procedure Q with parameters a and b is integrated in procedure P from a call Q(x,x), then IsAliased(a,b) is true. If Q is integrated in procedure P from a call Q(x,y), then IsAliased(x,y) is assigned to IsAliased(a,b). In Sections 6.2 and 6.3, we discuss two techniques for refining the aliasing information as other analysis proceeds.

In the worst case, every variable in the original program may be aliased to every other variable. Each assignment statement in such a program would be followed by V if - then structures, swelling the program by a factor proportional to the number of variables. If the number of aliases is large, it is correct (although pessimistic) simply to assign $\perp$ to any variables involved in a large number of alias pairs and not insert the if - then structures. (Pointers can also be modeled as assignments from $\perp$.)

6.2 Procedure Integration

The constant propagation techniques we have discussed seem to be well suited to performing some of the specializations needed when procedures are integrated. The specialized routines thus derived may be considerably smaller, as well as more efficient, than their generic progenitors. Moreover, we show how the techniques used in the SCC algorithm cause constant propagation to take time linear in the size of the resulting specialized procedure rather than the size of the generic one.

SCC has an advantage over SSC because the SCC algorithm can skip sections of the program that are inaccessible at execution time. In the compilers for languages such as Ada, C + +, and PL.8 where procedure integration is performed, or in compilers for variants of C or LISP where macro expansion is performed, or when code is created by a program generator, this may provide a significant improvement in both compile-time and execution-time performance.

Aliasing information can be improved by performing procedure integration. The aliasing information for a procedure is computed on the conservative assumption that the procedure can be called from any of its call sites. Any procedure that has been integrated can, by definition, be called from only one call site. The number of aliases passed through this one call site is typically less than the number passed through a common instance of the procedure. Once a procedure has been integrated, the parameter binding through that call site can be determined and many of the potential aliases can be ignored.

The benefit derived by combining constant propagation and procedure integration varies depending on the style of programming, the size of the program being compiled, the level of programming language being used, the other optimizations performed by the compiler, and how well those optimizations are integrated. If the programming style is to perform many optimizations by hand (such as procedure integration and constant propagation), many of the opportunities the compiler would have found may well have already been taken. If, however, the program is large, or has been frequently modified, then the programmer often loses track of opportunities for optimization. Higher-level languages, for example, LISP, ML, and SETL, inhibit the programmer from making too many low-level decisions. In languages such as C, FORTRAN, or Pascal, however, the programmer is free, and often required, to make many low-level decisions. This allows a programmer to write programs in such a way that the optimizer finds few opportunities for optimization other than those associated with arrays or register allocation. If the input to the compiler is fairly high level, a compiler that implements an aggressive set of well integrated optimizations is likely to find more opportunities to improve the program than one that does not.

It is therefore difficult to say whether one optimization is good or bad, and it is unlikely that any single study will provide a definitive result. Reports in the area are mixed.

Allen et al. [4], looking at a large sample of programs, have found that over 24 percent of all parameters to subroutines in PL/I are lexically constants. Presumably any global constant propagation algorithm would find additional ones. Moreover, in languages that make heavy use of generics or where runtime type information is needed, constant propagation has even more potential to be helpful.

Scheifler [39] studied procedure integration in CLU but did not consider performing any subsequent optimizations. He reported that procedure integration alone improved the performance of the resulting program, but at a considerable expense in the program size.

On the positive side, Ball [7] considered the effects of procedure integration on optimization, doing a form of ad hoc data flow analysis to estimate the effects of passing a constant parameter to a procedure. He reports positive results on the size and execution time of the compiled code, even though his constant propagation technique discovers only the simple constants. The ECS compiler at IBM [4] by design relied heavily on the use of procedure integration and subsequent tailoring of the code. Appel and Tim [6] used procedure integration (called beta-reduction in the LISP community) along with many optimizations in their ML compiler, and they report positive results.

On the negative side, Richardson and Ganapathi [35] studied the effects of optimization and procedure integration on five programs averaging 1200 lines each. They found that both procedure integration and optimization were of benefit separately, but their combined benefit was in general no more than the product of their separate benefits.

6.2.1 *The Procedure Integration Algorithm.* Procedure integration can be combined with constant propagation to achieve a better result than either separately. A prepass can create the SSA graph for each procedure. We can integrate only those statements that are executable based on constant propagation through the SSA graph of the procedure.

Time and space are saved by combining procedure integration with constant propagation rather than the more naive approach of integrating and then doing constant propagation. If one integrates first, one must expend both time and space in copying parts of the code not relevant for that execution, and then these irrelevant parts must be thrown away. In many compilers that perform procedure integration, the data flow information is not collected until after the integration.

The space explosion can be controlled by limiting the number of procedures that are inlined. There are three general strategies for choosing what to inline:

—Compiler directives. The programmer chooses some procedures that are inlined by the compiler or a preprocessor.

—Static analysis. The compiler analyzes the program and chooses some set of procedures to integrate. Common choices include integrating only leaf procedures or procedure calls nested within loops.

—Performance measurement. The program is compiled and run using some test data. Instrumentation is added to the program to count either the number of times each procedure is called or the number of times each procedure call is made.[7]

[7] The latter produces more precise information, since a single procedure may be called from many different locations and the frequency of each call may be different.

The first two strategies are easy to implement but do not generally inline the best set of procedures. The last produces better results but is cumbersome, since good test data is required and the program must be compiled several times.

Some modifications to SCC are necessary to perform procedure integration as described above, since procedure integration requires copying the code rather than simply marking it as executable. As the code is copied, edges must be updated so that the branches in the copied code point into the copied code rather than into the uninstantiated internal representation of the subroutine.

The problem is how to determine whether a node has been instantiated and, if so, where it resides. The solution is a global hash table that contains one entry for each instantiated node. The key in the hash table is constructed by concatenating two items: the name of the call site from which the procedure is being instantiated and the name of the node in the uninstantiated version (the place from which the code is being copied). The data in the hash table is the location at which the node was instantiated. Each newly instantiated node needs to record the name of the call site so that its branches can in turn be looked up in the hash table.

When a control flow graph edge is taken from the FlowWorkList, its target is looked up in the hash table. If a match is found, the algorithm looks at the data field to find the instantiated block. If a match if not found, the block has not yet been instantiated, and it must then be instantiated and added to the hash table. The hash table replaces the ExecutableFlag found in the single-procedure version of SCC.

SCC requires that all LatticeCells be initialized to $\top$ and that all ExecutableFlags are initialized to `false`. A naive implementation would require a pass over the entire procedure (even parts that would not be integrated) to perform this initialization. Instead, each LatticeCell and ExecutableFlag can be initialized as their defining nodes are copied. LatticeCells also reside in the hash table indexed by their name and the name of the instantiating call site.

6.3 Interprocedural Constant Propagation

The algorithm described in Section 6.2 can be viewed as a good, but sometimes unreasonably expensive, form of interprocedural constant propagation. The expense is a result of making explicit all possible paths through the program by expanding all of the procedures inline. If the program has recursive procedures, then the inlining process can be unbounded; otherwise, it may be exponential.

Where no procedure integration is performed, a variant of the procedure integration algorithm can perform interprocedural constant propagation. The statements in the procedure being integrated are not copied but are simply marked as being executable. This marking indicates that the statement may be executable along some path in the full program. Each call site to a given procedure jumps to the location where the procedure starts. The aliasing information at the entrance of the procedure is the meet of the aliasing information at each of the call sites. In effect, the SSA graph for the entire program looks like the interprocedural def-use chains described by Allen in [3].

This algorithm has the advantage that the amount of work required for constant propagation over the entire program is linear in the sum of the sizes of the SSA graph for each procedure; however, it has the disadvantage that the number of constants found may be small, since the only constants propagated across a procedure boundary are those having the same constant value at all call sites. This algorithm finds a class of aliases closely related to those found by Banning [8]. Our algorithm is more precise because it takes advantage of knowledge discovered during constant propagation.

Myers [29] describes two frameworks for performing several forms of interprocedural analysis. In his flow-sensitive framework, all paths through all procedures are examined. It is not surprising that the problems he investigated are intractable in this framework. In his flow-insensitive framework, the data-flow information for each procedure is summarized and the interprocedural information is created from this summary information and the call graph of the program. While Myers did not consider constant propagation, he did consider alias analysis. Our algorithm discovers many facts that, in Myers' terms, require flow-sensitive analysis, and yet the cost of our algorithm is linear in the size of the full program. Our algorithm is more precise because it takes advantage of knowledge discovered during constant propagation.

Two algorithms for interprocedural constant propagation have been published based on research done around the time of our conference paper [44]. The works by Callahan, Cooper, Kennedy, and Torczon [12], and Burke and Cytron [11] are motivated by two concerns: The amount of space may still be prohibitively large, since the entire program must be in memory at one time for the analysis, and any change in one procedure may force re-analysis of the entire program.[8] The algorithms in [11] and [12] can both be broken down into four steps. The details of each step differ between the two algorithms and the interested reader should see each paper for further information.

(1) Compute the *inter*procedural aliasing information.

(2) Compute the *intra*procedural constants.

(3) Summarize each procedure to determine the effects of *inter*procedural constants.

(4) Propagate the *inter*procedural constants along the paths in the call graph.

The novel aspect of these algorithms is how they summarize the functions and propagate the summarized results. While they address the time and space problems of our algorithm, they miss many of the constants that our

[8] Some people believe that even with large virtual memories, a worklist algorithm such as ours with fairly random access patterns over a large space will perform poorly. No hard data is available to resolve this.

algorithm finds. In addition, these algorithms are primarily intended for use in FORTRAN compilers, a program base in which data abstraction is rarely used; many of the interprocedural constants found by our algorithm are introduced by the use of data abstraction.

The weakness of the algorithms in [11, 12] is that there is no feedback between the steps. Discovering that certain values are constant may allow the discovery that some call is not executed. Discovering that some call is not executed may improve the aliasing information (yield fewer aliases into the called procedure) and may make more constants available into that routine, since the parameters are joined along fewer paths.

7. AREAS FOR FUTURE RESEARCH

In Figure 14, a very simple form of type determination was performed. In LISP, it is thought sufficient to propagate the type of any assignment node forward: The information can be propagated in the same way as in the constant propagation problem. To get good information in SETL, the problem is somewhat harder. The goal is to infer the type of the object from the way in which it is used (this problem was originally defined by Tenenbaum [41]). SETL has only one primitive data type to program with, the set. Since sets are rather inefficient to implement, the SETL compiler attempts to pick a more efficient representation of an object on the basis of the way the object is used.

The problem is bidirectional, since the information about how a variable is used must be propagated backward as well as forward. Tenenbaum's algorithm for doing this requires alternating forward and backward passes, with each pass done in a method similar to that of Kildall [26]. It may be possible to use a variation of the SSA graph to represent the propagation space. The chains must be bidirectional, that is, they must contain an edge from the use to the definition in addition to an edge from the definition to the use. There are many other details to be worked out, but this appears to be a straightforward extension of the ideas presented here.

In the *range propagation problem* [22, 24], the goal is to propagate ranges of values in an attempt to fix the upper and lower bounds of variables, so as to remove subscript range checking from areas of programs that can be proven safe. This problem differs from simple constant propagation in that the lattice may have an infinite number of levels, rather than just three. There are, however, subsets of this problem in which the number of lattice levels is small. For instance, in the problem of determining the possible values of a label variable in FORTRAN, the number of labels in a program is small and fixed. This type of problem should be easily solved by modifications to the algorithms presented here.

Arrays are difficult for almost any data flow analysis problem. The simple solution that is used in almost all implementations of optimizing compilers is to treat any assignment to an array as an assignment of $\perp$ unless that array is always indexed by constants. It may be possible to do something more sophisticated.

It is almost always desirable to integrate a function if all of the parameters are constant and the function references no global or free variables. Where only some parameters are constant, however, the decision is not so clear. Some benefit can be gained by unrolling loops and recursion if the space and time can be controlled by good heuristics. This problem has been investigated by Wegbreit [42], Ershov [18], Wegman [43], Appel and Jim [6], and Richardson and Ganapathi [36].

In this paper we have managed to combine constant propagation with unreachable code elimination and procedure integration. One of the open questions in compiler optimization is the proper order in which to apply the various optimizations. Some optimizations expose opportunities for other optimization techniques. We have eliminated the need to be concerned about the order of the optimizations we have combined and in the process have created a more powerful algorithm. It would be interesting to see if other techniques can be integrated in a similar manner.

8. CONCLUSIONS

The work presented here is based on three fundamental results concerning constant propagation: Kildall's definition of the problem involving the three-layered lattice, Reif and Lewis's algorithm involving a sparse representation of propagation space, and Wegbreit's algorithm, which used the result of conditional operations to improve the class of constants found. We have added two relevant results of our own. The first result is that a careful ordering of propagation in concert with symbolic execution of the conditional expressions can increase the number of constants found with no penalty in time or space. The second is the use of static single assignment form for constant propagation.

We have used these five results to craft an algorithm that is efficient in both time and space, and yet finds a very broad class of constants. Moreover, we have shown how to use this algorithm to perform procedure integration and interprocedural analysis.

ACKNOWLEDGMENTS

We would like to thank our colleagues Fran Allen, Trina Avery, Jeff Barth, Michael Burke, Larry Carter, Keith Cooper, Ron Cytron, Bill Harrison, Julian Padget, John Reif, Randy Scarborough, G. A. Venkatesh, and Referee B for all the help they have given us. We would particularly like to thank Barry Rosen who read the paper several times, always with helpful and important suggestions. This paper could not have been written without their assistance.

REFERENCES

1. AHO, A. V., SETHI, R , AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
2. ALLEN, F. E. A catalogue of optimizing transformations. In *Design and Optimization of Compilers.* R. Rustin, Ed., Prentice Hall, Englewood Cliffs, N.J , 1972, pp. 1-30.

3. Allen, F. E. Interprocedural data flow analysis. *Inf. Proc. 74* (1974), 398–402.
4. Allen, F. E., Carter, J. L., Fabri, J., Ferrante, J., Harrison, W. H., Loewner, P. G., and Trevillyan, L. H. The experimental compiling system. *IBM J. Res. Dev. 24*, 6 (Nov. 1980), 695–715.
5. Alpern, B., Wegman, M. N., and Zadeck, F. K. Detecting equality of values in programs. In *Conference Recordings of the Fifteenth ACM Symposium on Principles of Programming Languages*. (Jan. 1988), pp. 1–11.
6. Appel, A. W., and Jim, T. Continuation-passing, closure-passing style. In *Conference Recordings of the Sixteenth ACM Symposium on Principles of Programming Languages*. (Jan. 1989), pp. 293–302.
7. Ball, J. E. Predicting the effects of optimization on a procedure body. In *Proceedings of the SIGPLAN'79 Symposium on Compiler Construction*. (Aug. 1979), pp. 214–220. Published as *SIGPLAN Not. 14*, 8.
8. Banning, J. B. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Recordings of the Sixth ACM Symposium on Principles of Programming Languages*. (Jan. 1979), pp. 29–41.
9. Barth, J. M. An interprocedural data flow analysis algorithm. In *Conference Recordings of the Fourth ACM Symposium on Principles of Programming Languages*. (Jan. 1977), pp. 119–131.
10. Burke, M. An interval approach toward interprocedural analysis. Tech. Rep. RC 10640 47724, IBM, July 1984.
11. Burke, M., and Cytron, R. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*. (June 1986), pp. 162–175. Published as *SIGPLAN Not. 21*, 7.
12. Callahan, D., Cooper, K. D., Kennedy, K. W., and Torczon, L. M. Interprocedural constant propagation. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*. (June 1986), pp. 152–161. Published as *SIGPLAN Not. 21*, 7.
13. Chow, F. C. A portable machine-independent global optimizer—Design and measurements. Tech. Rep. 83-254 (Ph.D. thesis), Computer Systems Laboratory, Stanford Univ., Palo Alto, Calif., Dec. 1983.
14. Cocke, J. and Schwartz, T. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York Univ., April 1970.
15. Cooper, K. D. Interprocedural data flow analysis in a programming environment. Ph.D. thesis, Dept. of Mathematical Sciences, Rice Univ., 1983.
16. Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. Tech. Rep. RC 14756, IBM, revised Mar. 1991.
17. Ellis, J. R. Bulldog: A compiler for VLIW architectures. Ph.D. thesis, Dept. of Computer Science, Yale Univ., New Haven, Conn., Feb. 1985.
18. Ershov, A. P. On the essence of compilation. In *IFIP Working Conference on Formal Description of Programming Concepts*. (Aug. 1977).
19. Ferrante, J. and Ottenstein, K. J. A program form based on data dependency in predicate regions. In *Conference Recordings of the Tenth ACM Symposium on Principles of Programming Languages*. (Jan. 1983).
20. Furtney, M. and Pratt, T. W. Kernel-control tailoring of sequential programs for parallel execution. In *Proceedings of the 1982 International Conference on Parallel Processing*. (Aug. 1982), pp. 245–247.
21. Graham, S. L. and Wegman, M. N. A fast and usually linear algorithm for global flow analysis. *J. ACM 23*, 1 (Jan. 1976), 172–202.
22. Harrison, W. H. Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng. SE-3*, 3 (May 1977), 243–250.
23. Holley, L. H. and Rosen, B. K. Qualified data flow problems. *IEEE Trans. Softw. Eng. SE-7*, 1 (Jan. 1981), 60–78.
24. Johnson, H. Dataflow analysis for intractable systems software. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*. (June 1986), pp. 109–117. Published as *SIGPLAN Not. 21*, 7.

25. KAM, J. B. AND ULLMAN, J. D. Monotone data flow analysis frameworks. *Acta Inf. 7* (1977), 305-317.
26. KILDALL, G. A. A unified approach to global program optimization. In *Conference Recordings of the First ACM Symposium on Principles of Programming Languages.* (Oct. 1973), pp. 194-206.
27. MOREL, E. AND RENVOISE, C. Global optimization by suppression of partial redundancies. *Commun. ACM 22*, 2 (Feb. 1979), 96-103.
28. MUCHNICK, S. S. AND JONES, N. D, Eds. *Program Flow Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1981.
29. MYERS, E. W. A precise interprocedural data flow algorithm. In *Conference Recordings of the Eighth ACM Symposium on Principles of Programming Languages.* (Jan. 1981), pp. 219-230.
30. OTTENSTEIN, K. J. Data-flow graphs as an intermediate form. Ph.D. thesis, Dept. of Computer Science, Purdue Univ., Aug. 1978.
31. PRATT, T. W. Program analysis and optimization through kernel-control decomposition. *Acta Inf. 9* (1978), 195-216.
32. REIF, J. H. AND LEWIS, H R Symbolic evaluation and the global value graph. In *Conference Recordings of the Fourth ACM Symposium on Principles of Programming Languages.* (Jan. 1977), pp. 104-118.
33. REIF, J. H. AND LEWIS, H. R. Efficient symbolic analysis of programs *J. Comput. Syst. Sci. 32*, 3 (June 1986), 280-313.
34. REIF, J. H. AND TARJAN, R. E. Symbolic program analysis in almost linear time. *SIAM J. Comput. 11*, 1 (Feb. 1982), 81-93.
35. RICHARDSON, S. AND GANAPATHI, M. Interprocedural analysis vs. procedure integration. *Inf. Process. Lett.* 32, 3 (Aug. 1989), 137-142.
36. RICHARDSON, S. AND GANAPATHI, M. Code optimization across procedures. *IEEE Comput. 22*, 2 (Feb. 1989), 42-51.
37. ROSEN, B. K. Data flow analysis for procedural languages. *J. ACM 26*, 2 (April 1979), 322-344.
38. ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Global value numbers and redundant computations. In *Conference Recordings of the Fifteenth ACM Symposium on Principles of Programming Languages.* (Jan. 1988), pp. 12-27.
39. SCHEIFLER, R. W. An analysis of inline substitution for a structured programming language. *Commun. ACM 20*, 9 (Sept. 1977), 647-654.
40. SHAPIRO, R. M. AND SAINT, H. The representation of algorithms. Tech. Rep. CA-7002-1432, Massachusetts Computer Associates, Feb. 1970.
41 TENENBAUM, A. M. Type determination for very high level languages. Ph.D. thesis, Courant Institute of Mathematical Sciences, New York Univ., Oct. 1974.
42. WEGBREIT, B. Property extraction in well-founded property sets *IEEE Trans. Softw. Eng SE-1*, 3 (Sept 1975), 270-285.
43. WEGMAN, M. N. General and efficient methods for global code improvement. Ph.D. thesis, Computer Science Dept., Univ. of California at Berkeley, Berkeley, 1981.
44. WEGMAN, M. N. AND ZADECK, F. K. Constant propagation with conditional branches. In *Conference Recordings of the Twelfth ACM Symposium on Principles of Programming Languages.* (Jan. 1985), pp. 291-299.

Received February 1988; revised June 1989; accepted October 1990