Chapter 7

Value Numbering

Bowen Alpern
Kenneth Zadeck

December 28, 1992

7.1 Introduction

Value numbering is a technique for determining when two computations in a program are equivalent and eliminating one of them with a semantics-preserving transformation. Symbolic values are associated with computations in such a way that two computations are assigned the same symbolic value only if they are equivalent, that is, if they always compute the same value.

The problem of determining that two computations are equivalent is, in general, undecidable. A *conservative* solution to the equivalence problem—one that is correct, but not necessarily optimal—is required. In the context of value numbering, conservative means that any two expressions identified as equivalent produce identical values on all possible executions of the program, but that some equivalent computations may not be identified

as such.

Value numbering often discovers facts that constant propagation (Chap. 6) and redundancy elimination (Chap. 8) also discover. But there are also cases in which each technique discovers facts that the others miss, as shown in Figs. 7.1—7.3. Thus, the three techniques are incomparable; each may detect some opportunities for optimization that the others miss.

```
A ← 5
B ← A + 2
C ← A + 1
D ← C + 1
```

Figure 7.1. Constant propagation detects that both B and D are equal to 7. Value numbering misses this because it does not understand math.

Value numbering is both less and more powerful than constant propagation. Because constant propagation interprets function symbols, it can detect some equivalences that value numbering cannot. In Fig. 7.1, constant propagation would detect that B and D are both 7 and are therefore equivalent; value numbering would not detect the equivalence of B and D since it does not understand the meaning of the plus operator. Second, constant propagation discovers that an expression is equivalent to a constant, while value numbering may determine that two expressions are equivalent even if their values are not known at compile time. In Fig. 7.2, value numbering can determine that B is equivalent to D even though the values of these variables are not constant.

```
read A
B ← A + 2
C ← A
D ← C + 2
```

Figure 7.2. Value numbering detects that B is equal to D. Both constant propagation and redundancy elimination miss this.

Similarly, value numbering is both less and more powerful than redundancy elimination. First, two computations can be redundant even if they do not compute the same value. In Fig. 7.3, value numbering presumes that each of the three computations of X + 1 is different since at run time the computation in (3) is guaranteed to be equivalent to only one of (1) or (2). Redundancy elimination can remove the computation in line (3). On

the other hand, redundancy elimination normally requires that candidates for elimination be lexicographically identical. As illustrated in Fig. 7.2, value numbering does not make this requirement.

```
                      if ()
(1)                       then do read X; A ← X + 1; end
(2)                       else do read X; B ← X + 1; end
(3)                   C ← X + 1
```

Figure 7.3. Redundancy elimination detects that the expression in the assignment to C is redundant with the expressions in the assignments to A and B. Value numbering cannot detect this, since C is not always equal to A nor to B.

In general, the more resources (cycles and memory) devoted to an undecidable problem, the better the possible approximation. The easiest equivalence algorithm assumes that all expressions compute different values. This is cheap and conservative, but the results are somewhat disappointing. This chapter investigates more aggressive value-numbering techniques efficient enough for a production compiler.

Most of this chapter assumes that programs are in static single assignment (SSA) form. Section 7.2 defines a simple extension to SSA form and the notion of congruence: two variables are congruent only if they represent equivalent computations. Section 7.3 presents two techniques for detecting when expressions are congruent, the first optimistic and the second pessimistic. The pessimistic approach has a somewhat faster implementation, but does not handle loops well. These two approaches detect the same equivalences on programs without loops. Section 7.4 explains how to use congruence to improve a program. Section 7.5 describes some weaker value-numbering techniques commonly used on programs not in SSA form, and Sec. 7.6 gives historical notes.

7.2 Congruence of Variables

To discover equivalent computations, the variables of a program are *partitioned* into *congruence classes*. Intuitively, each variable in a congruence class has the same value. One expects that application of identical function symbols to equivalent arguments will yield equivalent results. Unfortunately, this is not true for programs in SSA form (Sec. 4.6.2): two ϕ-functions with identical arguments can produce different results if the ϕ-functions occur at different join points (see Fig. 7.4).

```
if P
    then A₁  ←  5
    else A₂  ←  3
A₃  ←  φ(A₁, A₂)
if Q
    then B₁  ←  5
    else B₂  ←  3
B₃  ←  φ(B₁, B₂)
```

Figure 7.4. Although the ϕ-functions defining A_3 and B_3 are identical, the results differ if P is not equal to Q.

For value numbering, it is convenient to modify SSA form slightly to ensure that identical functions with equivalent arguments produce equivalent results. In standard SSA form, each join point is followed by a sequence of assignments with ϕ-functions on their right-hand sides. Two ϕ-functions with the same arguments *that follow the same join point* produce identical results. Modified SSA form uniquely names each join point and subscripts each ϕ-function with the name of the corresponding join point.

In Modified SSA form (as in SSA form), each computation is represented as an assignment to a unique variable. Furthermore, each variable has a unique defining assignment. Thus there is a duality between the variables and the computations of a program. The function symbol on the righthand side of the defining assignment of a variable X is called the *operator* of X; the ith argument of that function symbol is called the ith *operand* of X.

A partitioning of these variables is said to be a *congruence partitioning* if and only if, for every partition P,

1. all the variables in P have the same operators (with the same arity);

2. the corresponding operands of these variables are in the same partitions.

Two variables X and Y are *congruent* (written X $\cong$ Y) if and only if they belong to the same partition of some congruence partitioning. One can show that if X and Y are congruent and the defining assignment of X dominates the defining assignment of Y, then whenever Y is used X has the same value as Y. Thus, Y can be eliminated from the program.

7.3 Determining Congruence

This section presents two techniques for partitioning the variables of a program in Modified SSA form into congruence classes. The *optimistic* technique starts from the assumption that all variables with the same operators (and arity) are in the same congruence class. These classes are partitioned further as contradictions to this assumption are uncovered. The *pessimistic* technique starts from the assumption that no variables are congruent, and variables are lumped together as new congruences are discovered.

Both techniques discover all congruences of loop-free programs in modified SSA form. For programs with loops, the optimistic technique discovers some congruences that the pessimistic technique misses. The optimistic technique has an implementation based on finite-state machine minimization [Hop71] with $O(N \log N)$ worst-case running time, where N is the number of variables in the program. The pessimistic technique has an implementation based on hashing with $O(N)$ expected running time.

7.3.1 Optimistic Value Numbering

Optimistic value numbering assumes, until evidence is uncovered to the contrary, that any variables that plausibly could be congruent are congruent. Variables tentatively thought to be congruent are placed in the same partition. Evidence that two variables are not congruent is:

1. they have different operators (or different arities); or

2. they have corresponding operands in different partitions.

If an operand of a variable in partition P belongs to partition Q, then for congruence the corresponding operand of every variable in P should be in Q. If not, then P must be further partitioned. Partition Q is said to *split* P into two partitions for each i, $P\backslash_i Q$ containing those variables in P whose ith operand is in Q and $P/_i Q$ containing those variables in P whose ith operand isn't in Q. Q *properly splits* P (for some i) if neither $P\backslash_i Q$ nor $P/_i Q$ is empty.

Splitting is the basis of optimistic value numbering. Initially, the variables are partitioned on the basis of their operators; two variables start out in the same class if the function symbols on the right-hand sides of their defining assignments are the same. Then each class is used to split the other classes. When no further splitting is possible, all the variables in a class must be congruent.

A naive implementation of this approach is quadratic in the number of variables, but a better implementation is possible. Using a class Q to

split other classes requires time proportional to the number of uses of the variables in it. Notice that, when Q has been used once to split the other classes, if Q itself is split into Q_1 and Q_2 then any class properly split by Q_1 is also properly split by Q_2. It is thus necessary to consider only one of the two. By always picking the class with the fewest uses, only $O(N \log N)$ classes need be used to split other classes. This algorithm is shown in Fig. 7.5.

```
worklist  ←  ∅;
foreach (function symbol f) do
      C_f  ←  ∅;
      foreach (assignment of the form:  "X  ←  f(A, B); ") do
            C_f  ←  C_f ∪ {X};
      od;
      worklist  ←  worklist ∪ {C_f};
od;
while (worklist ≠ ∅) do
      delete D from worklist;
      foreach (class C properly split by D) do
            if (C ∈ worklist) then worklist  ←  worklist − {C} ∪ {C\_iD, C/_iD};
            elsif (|uses(C\_iD)| < |uses(C/_iD)|) then
                  worklist  ←  worklist ∪  {C\_iD};
            else worklist  ←  worklist ∪  {C/_iD};
            fi;
      od;
od;
```

Figure 7.5. Optimistic value-numbering algorithm.

It remains to describe the data structures that let a class Q do all its splitting in time proportional to the number of uses of the variables of Q. The data structure for a class includes a count of the number of variables in the class, a count of the number of uses of those variables and a list of pointers to the structures for these variables. The data structure for a variable includes a pointer to the structure for its class, a pointer to (the structure for) each of its operands, a count of the number of uses of the variable, and a list of pairs. The first element of each pair is a pointer to a use of the variable; the second is an integer indicating which operand of the use the variable is. A list of classes P properly split by Q can be formed by visiting (the structures for) all variables defined by assignments that use variables in Q. The corresponding list of classes $P\backslash_i Q$ can be formed at the same time. The structure for a class P can be updated to be a structure for the class $P/_i Q$ without visiting the structures for the assignments in $P/_i Q$.

7.3.2   Pessimistic Value Numbering

During execution of the optimistic algorithm, variables later found to be incongruent may be lumped into the same class. The pessimistic technique never puts variables that are not congruent into the same class.

For a loop-free program, the pessimistic technique considers the variables in some topological order of their defining assignments (in the control-flow graph). This ordering ensures that a variable is considered before any variables that depend on it. When a variable is considered, it is compared against all previously considered variables. A variable X with defining assignment X $\leftarrow$ f(A,B) is placed in the same congruence class as a previously considered variable Y with defining assignment Y $\leftarrow$ g(C,D) if and only if f and g are identical, A and C have already been found congruent, and B and D have already been found congruent.

For programs with loops, there is no ordering by which each variable can be considered before those variables that depend on it, since there are cyclic dependencies among variables. The best one can do is to identify the back edges in the control flow graph and produce a topological ordering on the control-flow graph without the back edges. Reverse postorder (Sec. 2.2.2) may be used. Considering variables in this order breaks the cyclic dependencies. Since we know nothing about the values that enter the ϕ-function along the back edges, we must make conservative, i.e. pessimistic, assumptions: each ϕ-function at a node that is the target of a back edge produces a unique value that is *never* merged. To handle programs with loops, the pessimistic technique starts with each variable in a unique congruence class, which may be merged into a larger congruence class when the defining assignment for the variable is considered.

What congruences in programs with loops does the pessimistic approach fail to find? Both sets of assignments to X and Y in Fig. 7.6 are in fact congruent. The pessimistic approach cannot discover that the assignments to X and Y within the loop are congruent.

```
X  ←  1;  Y  ←  1
while(...)
    X  ←  X + 1
    Y  ←  Y + 1
end
```

Figure 7.6. Pessimestic value-numbering does not discover that X and Y are congruent; optimistic value numbering does.

The naive implementation of the pessimistic technique is quadratic in

the number of assignments in the program, but hashing can be used to implement it efficiently. Each congruence class is given a unique hash value called its *value number*. The value number of a variable X with defining assignment X ← f(A,B) is found by hashing the function symbol, f, combined with the value numbers of its operands.

The algorithm to implement this is shown in Fig. 7.7. Initially each defining assignment is assigned a unique value, so that no undefined values are present when ϕ-functions at the tops of loops are processed. The statements are then hashed[1] in reverse postorder. At the end of processing, all statements with identical hash numbers are congruent.

```
/*  We assume that the nodes of the CFG have been assigned  */
/*  reverse postorder numbers by DFS (Fig. 2.3.)  */
foreach (assignment of the form: "X ← f(A, B); ") do
    ValNum[X] ← UniqueValue();
od;
foreach (assignment of the form: "X ← f(A, B); "order rev.postorder) do
    ValNum[X] ← Hash(f||ValNum[A]||ValNum[B]);
od;
```

Figure 7.7. Pessimistic value-numbering algorithm.

7.4   Eliminating Congruent Assignments

Once the congruences are discovered, the program can be transformed to take advantage of them. The transformation is as follows: if variables V_1 and V_2 are congruent and the defining assignment of V_2 dominates the defining assignment of V_1, then the defining assignment to a variable V_1 is deleted and references to V_1 are replaced by references to V_2. Because programs contain conditional branches and joins, however, implementing this transformation is a nontrivial problem. This is similar to the problem of removing redundant computations described in Chap. 8, though rarely is so sophisticated a technique employed in the context of value numbering. Instead, a simple method, based on the dominator tree of the control flow graph, is used. In order to simplify its presentation below, we assume that a basic block contains only one statement.

As stated in Sec. 7.2, an expression assigning to V_1 in congruence class C can be replaced with V_2, also from congruence class C, if the location

---

[1] We assume that the hash function processes collisions so as to guarantee it never returns the same value for two inputs that are not the same.

of V_2 in the control flow graph dominates the location of V_1. A naive implementation of this test is quite expensive since the location of each assignment must be tested against all other locations of assignments in the same congruence class, and each test requires a traversal of the dominator tree. An efficient way to perform this test is shown in Fig. 7.5. By using the ancestor test for trees (described in Sec. 2.2.2) on the dominator tree and sorting the elements of the congruence class C by their location, each test can be done in constant time and the number of tests can be reduced to the size of C.

In this technique, associated with each variable is the $PreOrd$ number of the location of its defining assignment in the program's dominator tree. The variables in a congruence class are processed from lowest to highest $PreOrd$ number. The function $NextAssignment$ returns each of the elements in the congruence class in this order, and $ReplaceExpression$ changes the right-hand side of later expression to be a trivial assignment from the first variable. Copy propagation (Sec. 15.1.2) can then get rid of these trivial assignments.

$P \leftarrow NextAssignment()$;
$D \leftarrow NextAssignment()$;
while $(D \neq \emptyset)$ **do**
    if $(IsAncestor(P, D))$ **then call** $ReplaceExpression(D, P)$;
    else $P \leftarrow D$;
    fi;
    $D \leftarrow NextAssignment()$;
od;
procedure $ReplaceExpression(S1, S2)$
    foreach (use U of the target of $S1$) **do**
        rewrite U as a use of the target of $S2$;;
    od;
    delete statement $S1$;;
end $ReplaceExpression$

Figure 7.8. Algorithm to determine which congruences to delete.

7.5 Programs Not in SSA Form

In many compilers, value numbering is performed only over basic blocks or extended basic blocks. Easily accessible global-value numbering is a relatively new technique and all of the global techniques rely on an intermediate representation similar to SSA form. Since it may not be feasible

to add SSA form to an existing compiler, we give here a modified version of the value-numbering algorithm that does not require SSA form.

Two properties of SSA form are used by value numbering: first, ϕ-functions are located at the join points in the control-flow graph, and second, each variable has only a unique definition point. The ϕ-functions model the semantics of join points in a program. If the intermediate form does not have ϕ-functions, no information about the value numbers of the predecessors can be propagated past a join point. A value-numbering algorithm not based on SSA form must treat each join node in the same way as the pessimistic algorithm treats joins that are the destination of back edges. Since the optimistic technique offers no advantages in representations without ϕ-functions, we focus on modifications to the pessimistic technique. Essentially, variables used but not defined in an extended basic block are assumed not to be congruent to any other variables. All the uses of a variable in each extended basic block are initially given a unique value number.

Without ϕ-functions, value numbering can discover only congruences local to extended basic blocks. Figure 7.9 demonstrates this: in each arm of the conditional statement, W is found to be congruent to X, but because there are no ϕ-functions to join the values together, Y and Z are not found to be congruent.

```
if P
    then do W ← 1; X ← 1; end
    else do W ← 2; X ← 2; end
Y ← W + 3
Z ← X + 3
```

Figure 7.9. Without SSA form, each W is found congruent to the X in the same arm, but Y is not found congruent to Z.

If the SSA-form property that each variable on the left side of an assignment must have a unique name is not satisfied, certain equivalences may not be found. The value given to X in (1) on the left of Fig. 7.10 would be lost. For simple basic blocks, unique names for variables can be obtained by replacing all but the last assignment to each variable with assignments to uniquely named temporary variables. On the right of Fig. 7.10, the first assignment to X has been replaced by an assignment to T1, and the use of X in (2) must also be replaced by T1. The algorithm discovers that T1 and Y are congruent.

For extended basic blocks the problem is more difficult. There is, in general, no unique last assignment to each variable. On the left of Fig. 7.11,

```
(1)  X ← A + B              T1 ← A + B
(2)  X ← X + 1              X ← T1 + 1
(3)  Y ← A + B              Y ← A + B
```

Figure 7.10. On the left, without unique temporary variables, statement (2) kills the value computed by statement (1) and thus Y is found to be equal to the X at (1). Introducing the temporary T1 allows it to be found congruent to Y.

if P is true, the second assignment is the last assignment to W; if P is false, the first assignment is the last assignment to W. In order to value number this extended basic block properly, a store to W must be inserted along the control flow edge executed if P is false, as shown on the right of Fig. 7.11.

```
W ← 1                      T1 ← 1
if P                       if P
    then W ← W + 1             then W ← T1 + 1
                               else W ← T1
```

Figure 7.11. Adding unique temporaries is harder in code with branches.

The technique to identify the last store requires two passes over the tree of blocks in the extended basic block, the first pass bottom-up (all children before the parent) and the second top-down. The first pass builds a set, *FutureDefs*, that is associated with each basic block and contains the names of variables assigned to in that basic block or in one of the descendants of that block. The second pass inserts stores onto the paths that need them. A store for a variable V is needed at the top of a basic block *B* if V is not in *FutureDefs*(*B*) and V was assigned to in the predecessor of *B*.

The second pass also performs the renaming to change all the other statements into assignments to temporary variables. This renaming is accomplished by maintaining a symbol table for each basic block. Each use of a variable is looked up in the table and replaced with the name of the temporary variable last assigned to that variable. If no temporary name is found for that variable (i.e., the name was upwards-exposed on entry to the extended basic block), then the old variable name is used. Each assignment statement allocates a new temporary and replaces the entry in the table for that basic block. It is necessary to keep different tables for each block since assignments that occur in a sibling to basic block *B* should not affect

B. The table for basic block B is initialized from the table of $parent(B)$ after all of the assignments in the parent have occurred.

7.6   Historical Notes

Many compilers use the pessimistic algorithm over single basic blocks. This algorithm was originally proposed by Cocke and Schwartz [CS70] and was modified to work over extended basic blocks in the PL.8 compiler [AH82].[2]

The first global, optimistic algorithm was proposed by Reif and Lewis [RL77, Rei78, RL86]. Their algorithm has a slightly better asymptotic complexity than the technique presented here, but is difficult to implement.

Using Hopcroft's finite-state minimization algorithm to determine the congruences was first suggested by Alpern, Wegman, and Zadeck [AWZ88]. This paper also proposed some stronger subscripting schemes for the ϕ-functions that capture the semantics of conditionals and loops and have been extended by Yang, Horowitz, and Reps [YHR89] to solve problems that arise in trying to automatically merge different versions of programs. Recently Ballance, Maccabe and Ottenstein [BMO90] have combined SSA form with the program dependence graph (PDG) and have called the result *gated single assignment* (GSA) form. This may prove to be the basis of a better subscripting scheme than that proposed in [AWZ88].

The value-numbering algorithm of Sec. 7.5 discovers exactly those equivalences that Reif and Lewis's algorithm would have discovered if each ϕ-function were labeled with the name of the variable (in the original program) as well as the name of the join point. It is natural to ask if weaker labeling of the ϕ-functions would allow more equivalences to be discovered. The answer is yes; in [AWZ88] the ϕ-functions for join points of conditional statements are labeled with the value number of the predicate for the conditional statement. Other labeling schemes are possible. All that is required is that ϕ-functions with identical labels and equivalent arguments must produce equivalent results.

Acknowledgments

---

[2] The modifications were, in fact, quite extensive because of the data structures used to represent the intermediate code; see Chap. 18 for details.

Bibliography

[AH82]   M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, pages 22–31, June 1982. Published as SIGPLAN Notices Vol. 17, No. 6.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 1–11, January 1988.

[BMO90] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web. *Proc. SIGPLAN'90 Symp. on Compiler Construction*, pages 257–271, June 1990. Published as SIGPLAN Notices Vol. 25, No. 6.

[CS70]   J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York U., April 1970.

[Hop71]   J. Hopcroft. An n log n algorithm for minimizing the states of a finite automaton. *The Theory of Machines and Computations*, pages 189–196, 1971.

[Rei78]   J. R. Reif. Symbolic interpretation in almost linear time. *Conf. Rec. Fifth ACM Symp. on Principles of Programming Languages*, pages 76–83, January 1978.

[RL77]   J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. *Conf. Rec. Fourth ACM Symp. on Principles of Programming Languages*, pages 104–118, January 1977.

[RL86]   J. H. Reif and H. R. Lewis. Efficent symbolic analysis of programs. *J. Computer and System Sciences*, 32(3):280–313, June 1986.

[YHR89] W. Yang, S. Horwitz, and T. Reps. Detecting program compo-
         nents with equivalent behaviors. Technical Report 840, Dept. of
         Computer Sci., U. of Wisconsin at Madison, April 1989.