A Simple Algorithm for Global Value Numbering

Nabizath Saleena and Vineeth Paleri

Department of Computer Science and Engineering

National Institute of Technology Calicut, India.

{saleena,vpaleri}@nitc.ac.in

Abstract

Global Value Numbering(GVN) is a method for detecting redundant computations in programs. Here, we introduce the problem of Global Value Numbering in its original form, as conceived by Kildall(1973), and present an algorithm which is a simpler variant of Kildall's. The algorithm uses the concept of *value expression* - an abstraction of a set of expressions - enabling a representation of the equivalence information which is compact and simple to manipulate.

1 Introduction

Detection and elimination of redundant computations have been interesting topics in the area of code optimization in compilers. Value Numbering originated as a method for detecting redundant computations within a basic block (known as Local Value Numbering). The basic idea is to assign a number to each expression in such a way that *equivalent* expressions are assigned the same number [2]. Two expressions are said to be *equivalent* if we can statically determine that both the expressions will have the same value during execution.

The problem of Global Value Numbering(GVN) is how to extend the idea of local value numbering to detect redundant computations globally, within a program. An initial attempt on GVN can be found in Kildall [4]. In the *structuring approach* of Kildall [4], for every expression in the program, Kildall computes and maintains all its equivalent expressions, leading to exponential sized partitions. Most of the works that followed, tried to make the process more efficient by means of special program representations and data structures [1, 3, 5, 6]. In general, we feel that there is a lack of clarity in the underlying concept of GVN and a lack of simplicity in the solutions.

In fact, in his implementation notes, Kildall suggested a *value numbering approach* to ensure linear sized partitions [4]. An expression with value numbers as operands, called *value expression*, is used to represent a set of expressions

that are equivalent. At program points where multiple control flow paths merge, the equivalence information is computed by means of a *confluence* operator. Confluence of equivalence classes involving *value expression*s is a little bit tricky, and may be due to this reason the method did not draw much attention. Here, we make use of the concept of *value expression* to devise a simple algorithm for GVN.

We start with notations and definitions in Section 2. The algorithm for global value numbering is given in Section 3. This is followed by some comments on the algorithm in Section 4 and conclusions in Section 5.

2   Notations and Definitions

2.1   Program Representation

The input to the algorithm is assumed to be a flow graph, with an empty entry node, denoted *entry*, and an exit node, denoted *exit*. Each node contains at most one assignment statement in three-address code[1]. Each assignment statement is of the form $x = e$, where x is a variable and e is an expression. An *expression* is either a constant, a variable, or an expression of the form $y \ op \ z$ where y and z are variables or constants and op is an operator. For a node n in the flow graph, the input and output points of the node are denoted by IN_n and OUT_n respectively.

2.2   Expression-pool

For a program point, the *expression-pool* at that point denotes the set of expressions that are equivalent at that point. This is represented as a partition of expressions into equivalence classes. Each class in the pool has a *value number*, denoted v_i, where i is a positive integer. For convenience, we show the *value number* of a class as the first element in it. As an example, $\{[v_1, \ a, \ x], \ [v_2, \ b, \ y]\}$ shows an *expression-pool* with two equivalence classes. The value number v_1 is assigned to a and x and value number v_2 is assigned to b and y. For a node n in the flow graph, we use EIN_n and $EOUT_n$ to denote the *expression-pool*s at IN_n and OUT_n respectively.

2.3   Value Expression

For each expression of the form $x \ op \ y$, we can obtain a *value expression*, by replacing the operands of the original expression by the corresponding *value number*s. For example, suppose the expression-pool $\{[v_1, \ a, \ x], \ [v_2, \ b, \ y]\}$ reaches a node containing the statement $z = x + y$. Here, the *value expression* of $x + y$ is $v_1 + v_2$. Instead of the program expression $x + y$, we put its *value expression* in the pool, with a new value number say v_3, to obtain the output pool, $\{[v_1, \ a, \ x], \ [v_2, \ b, \ y], \ [v_3, \ v_1 + v_2, \ z]\}$.

---

[1]For simplicity, we do not consider other statements.

We can see that the *value expression* $v_1 + v_2$, represents not just $x+y$, but the set of equivalent expressions $\{a+b,\ x+b,\ a+y,\ x+y\}$. The presence of v_1+v_2 in the pool indicates that an expression from this set is already computed, and this information is enough for detection of redundant computations. The interesting point to note is that a single binary *value expression* can represent equivalence among any number of expressions of any length. As an example, with respect to the above pool, the *value expression* $v_1 + v_3$ represents the expressions $a + z$, $x + z$, $a + (a + b)$, $a + (x + b)$, $x + (a + b)$, and so on.

3 Value Numbering: Algorithm

The algorithm computes the expression-pool at every program point. For a node n in the flow graph, if EIN_n is given, then we can compute $EOUT_n$ by means of the *transfer function* associated with node n. The points at which multiple control flow paths join are called *confluence points*. For computing the *expression-pool*s at such points, we define a *confluence operator*.

3.1 Transfer Function

The *transfer function* associated with a node n, denoted f_n, computes $EOUT_n$ using EIN_n, i.e. $EOUT_n = f_n(EIN_n)$. Algorithm 1 shows the transfer function for a node n containing an assignment $x = e$, where x is a program variable and e is an expression[2]. The assignment can be considered as *killing* all expressions involving the variable x and *generating* the new equivalence between x and e. The effect of *killing* expressions is achieved by removing x from its class, say C_i. Now, if C_i is a singleton with its value number, say v_i, as the only element in it, we delete the class C_i and any *value expression*s involving v_i from the pool. The function $deleteSingletons(E)$ is assumed to do these steps repeatedly till no singleton classes remain in the pool E.

The function $valueExp(e)$ returns the *value expression* of e, if e is of the form $x\ op\ y$, and returns e itself otherwise. If e is already assigned a value number in EIN_n, say v_e, then we put x in the same class as that of e. Otherwise, a new class containing x and e is created together with a distinct *value number* in it and this new class is added to the output pool (if e contains an operator, then instead of e, we add its value expression).

3.2 Confluence Operator

The *expression-pool* at a confluence point should contain the sets of equivalent expressions common to all incoming pools. The common expressions that are explicitly present in the input pools can be obtained by a simple class-wise intersection of *expression-pool*s. The hard part is obtaining the equivalence information based on the *value expression*s in the incoming pools. In the example

---

[2]if x occurs in e, we assume that the statement is split into $t = e$ followed by $x = t$ where t is a new distinct temporary variable.

Algorithm 1: Computes $EOUT_n = f_n(EIN_n)$, for a node n containing the assignment $x = e$.

$E_t = EIN_n$;
if (x is in a class $C_x \in E_t$)
    then remove x from C_x;
       $deleteSingletons(E_t)$;
$e' = valueExp(e)$;
if (e' is in a class $C_{e'} \in E_t$)
    then add x to $C_{e'}$;
    else create a new class C_k, with x and e' together with a new
       value number v_k in it, and add C_k to E_t;
$EOUT_n = E_t$;
return $EOUT_n$;

$E_1\colon \{[v_1,\ x,\ a],\ [v_2,\ y,\ b],\ [v_3,\ v_1 + v_2,\ z]\}$ $E_2\colon \{[v_4,\ x,\ c],\ [v_5,\ y,\ d],\ [v_6,\ v_4 + v_5,\ s]\}$

$E_3\colon \{[v_7,\ x\},\ [v_8,\ y],\ [v_9,\ v_7 + v_8]\}$
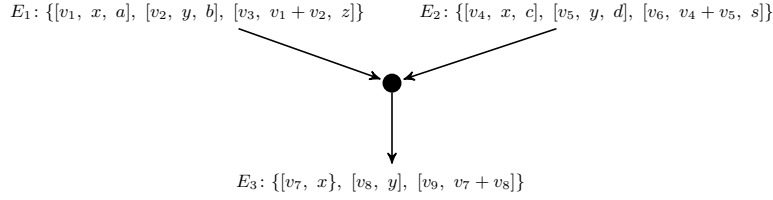
Figure 1: Computing confluence

shown in Figure 1, we see two expression-pools, E_1 and E_2, reaching a confluence point. Let E_3 be the pool resulting after confluence. Since x occurs in both the input pools E_1 and E_2, we put it in the output pool E_3. Since the value numbers of x are different in the input pools, we assign a new distinct value number v_7 for the resulting class. Similarly, we put y in E_3 with new value number v_8. In E_1, the *value expression* $v_1 + v_2$ represents the set of equivalent expressions $x + y$, $x + b$, $a + y$, and $a + b$. In E_2, the *value expression* $v_4 + v_5$ represents the set of equivalent expressions $x + y$, $x + d$, $c + y$, and $c + d$. Here, we see a common expression $x + y$ represented by $v_1 + v_2$ in E_1 and $v_4 + v_5$ in E_2. Let us now devise a method to collect such common expressions by examining the *value expressions* in the input pools.

Consider the corresponding operands of the pair of *value expressions* $v_1 + v_2$ and $v_4 + v_5$. We see a common element x in the classes of v_1 and v_4 and a common element y in the classes of v_2 and v_5. In other words, the intersection of the classes of v_1 and v_4 is non empty and also the intersection of the classes of v_2 and v_5 is non empty. This is enough to infer that there is a common expression represented by the two *value expressions*. At confluence, the intersection of the classes of v_1 and v_4 results in a class with value number v_7 and the intersection of the classes of v_2 and v_5 results in a class with value number v_8. Hence the common expression $x + y$, after confluence, gets the *value expression* $v_7 + v_8$

and this can be added to E_3.

In general, let there be a class with value number v_i and *value expression* $v_{i1}+v_{i2}$ in E_1, and let there be a class with value number v_j and *value expression* $v_{j1} + v_{j2}$ in E_2. If the intersection of the classes of v_{i1} and v_{j1} results in a non empty class with *value number* v_{k1}, and the intersection of the classes of v_{i2} and v_{j2} results in a non empty class with *value number* v_{k2}, then we can conclude that the pair of *value expression*s, $v_{i1} + v_{i2}$ and $v_{j1} + v_{j2}$, represent a common expression e in the two pools. The *value expression* of e after confluence is $v_{k1} + v_{k2}$ and this can be added to E_3.

An algorithm for computing the confluence of two expression-pools E_i and E_j is given in Algorithm 2. We use the symbol $\bigwedge$ to denote the confluence operation. The algorithm takes each pair of classes, $C_i \in E_i$ and $C_j \in E_j$, and finds the common expressions in C_i and C_j (either explicitly present or implicitly represented by *value expression*s). The operation of finding the

---

Algorithm 2: Computing confluence of expression-pools, E_i and E_j, i.e. $E_i \bigwedge E_j$.

$E_k = \Phi$;
foreach pair of classes, $C_i \in E_i$ and $C_j \in E_j$
    $C_k = C_i \sqcap C_j$;
    if $(C_k \neq \Phi)$
        then add C_k to E_k;
$deleteSingletons(E_k)$;
return E_k;

---

common expressions in C_i and C_j can be considered as a special intersection and we denote it as $C_i \sqcap C_j$. Algorithm 3 shows the computation of $C_i \sqcap C_j$.

Algorithm 3: Computing $C_i \sqcap C_j$.
Note: A class with value number v_n is denoted by C_n and vice-versa.

$C_k = \Phi$;
foreach $e \in C_i \cap C_j$
    add e to C_k;
if $(C_i$ and C_j have different *value expression*s$)$
    then
        // let $v_{i1} + v_{i2}$ and $v_{j1} + v_{j2}$ be the `value expressions`
        // in C_i and C_j respectively
        $C_{k1} = C_{i1} \sqcap C_{j1}$;
        $C_{k2} = C_{i2} \sqcap C_{j2}$;
        if $(C_{k1} \neq \Phi$ and $C_{k2} \neq \Phi)$
            then add the *value expression* $v_{k1} + v_{k2}$ to C_k;
if $(C_k \neq \Phi$ and C_k does not have a *value number*$)$
    then add a new value number, say v_k, to C_k;
return C_k;

3.3   The Algorithm

Algorithm 4 shows the main function for Global Value Numbering. T is the *top* element such that $E_i \bigwedge T = E_i$, for any expression-pool E_i. For a node n,

---

Algorithm 4: Computes EIN_n and $EOUT_n$ for each node n.

$EOUT_{entry} = \Phi$;
foreach node $n \neq entry$ **do** $EOUT_n = T$;
while (changes to any $EOUT$ occur) **do**
    // We mean changes in the equivalence information.
    // The changes only in value numbers can be ignored.
    foreach node $n \neq entry$ **do**
        $EIN_n = \bigwedge\limits_{p \in pred(n)} EOUT_p$;
        $EOUT_n = f_n(EIN_n)$;

$pred(n)$ denotes the set of immediate predecessors of n, and $\bigwedge\limits_{p \in pred(n)}$ computes the confluence of expression-pools that reach the output of its predecessors.

4   Comments on the Algorithm

Power of the Algorithm   Figure 2 shows an example of redundancy detection which will demonstrate the power of the algorithm, especially that of the confluence operation. Let us use C_i to denote a class with value number v_i. The

$$c = a + b$$
$$e = c + z$$
$$d = p + q$$
$$f = d + z$$

E_1: $\{[v_1,\ x,\ a],\ [v_2,\ y,\ b],\ [v_3,\ v_1 + v_2,\ c],$
$\quad\quad [v_4,\ z],\ [v_5,\ v_3 + v_4,\ e]\}$

E_2: $\{[v_6,\ p,\ x],\ [v_7,\ q,\ y],\ [v_4,\ z],$
$\quad\quad [v_8,\ v_6 + v_7,\ d],\ [v_9,\ v_8 + v_4,\ f]\}$

E_3: $\{[v_{10},\ x],\ [v_{11},\ y],\ [v_4,\ z], [v_{12},\ v_{10} + v_{11}],\ [v_{13},\ v_{12} + v_4]\}$
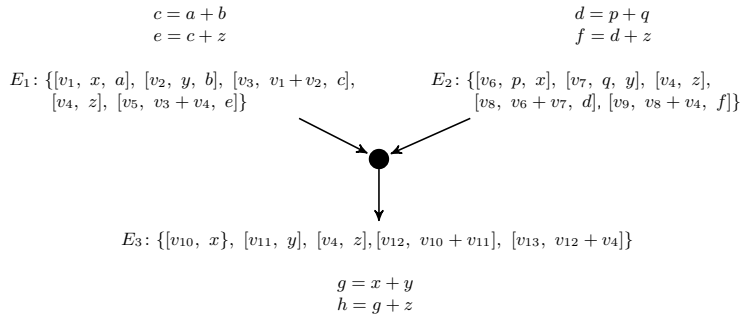
$$g = x + y$$
$$h = g + z$$

Figure 2: Value Numbering to detect redundancy

value expression $v_1 + v_2$ in C_3 and $v_6 + v_7$ in C_8 represent a common expression $x + y$. When we compute the confluence, $C_3 \sqcap C_8$ results in the class C_{12} and the *value expression* $v_{10} + v_{11}$ in it represents $x + y$. Another common expression is $(x + y) + z$, represented by the *value expression*s in C_5 and C_9. The operation $C_5 \sqcap C_9$, results in C_{13}, whose *value expression* $v_{12} + v_4$ represents $(x + y) + z$. After confluence, when we do value numbering of $g = x + y$, since $x + y$ maps to

6

$v_{10} + v_{11}$, a *value expression* in E_3, it is detected as redundant and g gets *value number* v_{12}. Similarly, the expression $g + z$ maps to $v_{12} + v_4$ and hence this is also detected as redundant.

A Comparison With Some of the GVN Algorithms   In terms of power, our algorithm is as precise as Kildall's approach[4]. Alpern, Wegman, and Zadeck's (AWZ) algorithm [1] is an efficient algorithm for GVN, but is not as precise as Kildall's. The AWZ algorithm fails to detect the category of equivalences shown in Figure 2. The algorithm given by Gulwani and Necula [3] does intersection of only those classes having at least one common variable. But as per our observation, intersection of all pairs of classes is required for detecting the kind of equivalences similar to that shown in Figure 2.

5   Conclusion

An algorithm for Global Value Numbering is presented. The concept of *value expression* enables a compact representation of equivalence and simplifies the computation of confluence. It may be noted that a single binary *value expression* can represent equivalence among any number of expressions of any length. We feel the algorithm is simpler compared to that available in the literature. In terms of power, it is as precise as Kildall's approach.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In 15th *ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[2] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2000.

[3] S. Gulwani and G.C. Necula. A polynomial time algorithm for global value numbering. *Science of Computer Programming*, 64(1):97–114, 2007.

[4] G.A. Kildall. A unified approach to global program optimization. In 1st *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[5] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In 15th *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[6] O. Ruthing, J. Knoop, and B. Steffen. Detecting equality of variables: Combining efficiency with precision. In 6th *International Symposium on Static Analysis*, pages 232–247, 1999.