Detecting Equality of Variables in Programs

Bowen Alpern
Mark N. Wegman
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
F. Kenneth Zadeck
Department of Computer Science
Brown University
Providence, RI 02912

1 Introduction

This paper presents an algorithm for detecting when two computations produce equivalent values. The equivalence of programs, and hence the equivalence of values, is in general undecidable. Thus, the best one can hope to do is to give an efficient algorithm that detects a large subclass of all the possible equivalences in a program.

Two variables are said to be *equivalent at a point* p if those variables contain the same values whenever control reaches p during any possible execution of the program. We will not examine all possible executions of the program. Instead, we will develop a static property called *congruence*. Congruence implies, but is not implied by, equivalence. Our approach is *conservative* in that any variables detected to be equivalent will in fact be equivalent, but not all equivalences are detected.

Previous work has shown how to apply a technique called *value numbering* in basic blocks [CS70]. Value numbering is essentially symbolic execution on straight-line programs (basic blocks). Symbolic execution implies that two expressions are assumed to be equal only when they consist of the same functions and the corresponding arguments of these functions are equal. An expression DAG is associated with each assignment statement. A hashing algorithm assigns a unique integer, the *value number*, to each different expression tree. Two variables that are assigned the same integer are guaranteed to be equivalent. After the code

```
B ← A + 3
C ← B * 5
D ← (A + 3) * 5
```

the value number of C and D is the hash value of "(A+3)*5".

Because calculation is done only symbolically, nothing can be said about variables with different value numbers. If the assignment to D had read

```
D ← (A * 5) + 15
```

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specfic permission.

© 1988 ACM-0-89791-252-7/88/0001/0001 $1.50

no value numbering algorithm would recognize that C and D are the same, since value numbering is based on symbolic computation. It is easy to generalize value numbering to extended basic blocks.

Reif and Lewis [RL77] have given a complex and efficient algorithm for detecting a somewhat more general form than extended basic blocks, but their algorithm cannot detect that in the following sequence J and K must be equal:

```
if P then J ← 5 else J ← 6
if P then K ← 5 else K ← 6
```

One contribution of this paper is to give an efficient algorithm for detecting equality in the presence of control structures, including `if-then-else` and loops. Instead of associating an expression DAG with every assignment, we associate nodes in a directed, cyclic value graph. Because cycles may be present, it is not clear how to represent values so that they may tested easily for equality. We therefore define the notion of *congruence* for nodes of the value graph. Determining which nodes are congruent is a partitioning problem an so is essentially the same problem as minimizing a finite-state machine, which Hopcroft [Hop71] has shown can be computed in $O(E \log E)$.

Intuitively, our algorithm will make a list of sets of variables it has discovered to be equivalent. Such a list is a *fixed point* if the variables associated with two expressions are in the same set when they have the same functions and when the list contains the information that their subexpressions are equal. When the program has loops, there can be a chicken-and-egg problem: if the results of two expressions are later used in the computations determining the value of their sub-expressions, it is not immediately clear how equality should be determined.

As with other flow analysis algorithms, we will find a *maximal fixed point* [Weg75] [GW76]. The maximal fixed point is a fixed point that contains the most equal values. The algorithms presented here are *optimistic*: they proceed by initially assuming that all values are equal and then separate them into more and more sets of possibly equal variables.

Another advantage of our algorithm is that it can be easily extended to exploit additional facts about program semantics. We will give two such extensions to the algorithm: others are clearly possible.

Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California (January 1988)

Uses of this algorithm include:

1. Register Allocation: There is no need to load a value if a register already contains a value which can be shown to be equal.
2. Common Sub-Expression Elimination: There is no need to recalculate a value if it has already been calculated. If an expression is calculated at a node u and later at a node **v**, if **u** dominates **v**, and if we can show that the variables are congruent at **v**, then the calculation at **v** need not be performed.
3. Movement of Invariant Code: Detection whether or not an expression will calculate the same value in a different location is a key concern.
4. Branch Elimination: If a value is guaranteed to be equal to another value, equality tests can be eliminated. This condition can be expected in code in which procedures have been integrated.
5. Branch Fusion and Loop Jamming: If there are two **if-then-else** statements one right after the other, and their predicates result in equal values, then the code on each branch of the second statement can be moved into the appropriate branch of the first and the second test can be eliminated. Similarly, if two loops have predicates we find equivalent, then the loops are executed the same number of times; if there are no dependencies, then the loops can be combined. Both of these program transformations may make further analysis easier and allow other optimizations to take place.

Section 2 presents the basic algorithm and shows how to use it when programs are viewed as flow graphs. Section 3 shows how to use program structure to discover additional equivalences. Section 4 presents more two generalizations of the basic algorithm which can be used to detect more equalities. We conclude in Section 5.

2 Equivalence of Variables in Simple Programs

Here we introduce the basic machinery used to detect equivalences. This will be the basis of more sophisticated algorithms in Sections 3 and 4. First we need to define further what it means for variables to be *equivalent*. Consider the following program text:

```
if Q
   then do
      I ← 5
      J ← 5
   end
   else do
      I ← 6
      J ← 7
   end
```

Are variables I and J equivalent? At the end of the **if-then-else** there are two possible answers; when viewed *dynamically*, the answer depends on the value of Q; when viewed *statically*, the answer must be **no** since we wish our algorithm to be conservative. However, at the end of the **then** block, the answer is **yes** even viewed statically.

We introduce the notion of *congruence* which is one of the two conditions that we need to detect equivalence. Congruence is a relationship between two variables without respect to location in the program. To make it possible to detect such equivalences at differenct points in the program, we introduce several new variables for each variable in the original program. Thus, in the above example, we will break I into three distinct variables: one in the **then** clause, one in the **else** clause, and the third at the end of the **if-then-else**. (This will be explained more fully in Section 2.2.) The I in the **then** clause will be congruent to the J in the **then** clause.

Even in the **then** clause, it is not clear what it means to say that I and J are equivalent. I equals J only after the assignment to J. We can assert that two variables are equivalent at a point **p** only if we know the assignments to both variables will have been executed whenever control reaches **p**. We can determine statically that the assignments have been executed if both assignments dominate **p**. A node **a** in a rooted, directed graph is said to dominate **b** if all paths from the root to **b** go through **a**. We will introduce auxiliary assignments (and auxiliary variables) so that each use of a variable is dominated by an assignment to the variable.

The algorithm can be broken into four steps:

1. Build a control flow graph that represents the program.
2. Replace each variable (both scaler and arrays) in the original program with several new variables. These new variables have the property that there is only one assignment to them in the program. This form is called *static single assignment* form, or SSA form.
3. Build an auxiliary structure called the *value graph* that represents the symbolic execution of the program. Label each of the assignments in the SSA program with a node in the value graph. (In Section 3 we will modify the value graph so that we can discover a larger set of congruences which are dependent on control structure.)
4. Determine congruence of nodes in the value structure. Two variables will be equivalent at a point **p** if their assignments dominate **p** and are labeled by congruent nodes. (In Section 4 we will modify the algorithm to detect further congruences.)

The example in Figure 1 will be used throughout this section.

2.1 Building the Control Flow Graph

We construct a *control flow graph* for the program and each node corresponds to a basic block in the program. Each edge in the control flow graph corresponds to a branch in the program. There may be multiple edges into a node. Such nodes are called *join points* and we assume the incoming edges are ordered. Such a construction is fairly standard in the study of compilers; see Allen [All70] for details.

```
if (I < 29)
   then do
      J ← 1
      K ← 1
   end
   else do
      J ← 2
      K ← 2
   end
if (I < 29)
   then L ← 1
   else L ← 2
```

Figure 1: A Simple Example

In Figure 2, each box represents a basic block; the edges connecting the boxes are the control flow graph edges and the numbers to the side of the boxes are the labels for each basic block. They can be assigned in any way that gives a unique name to each block.

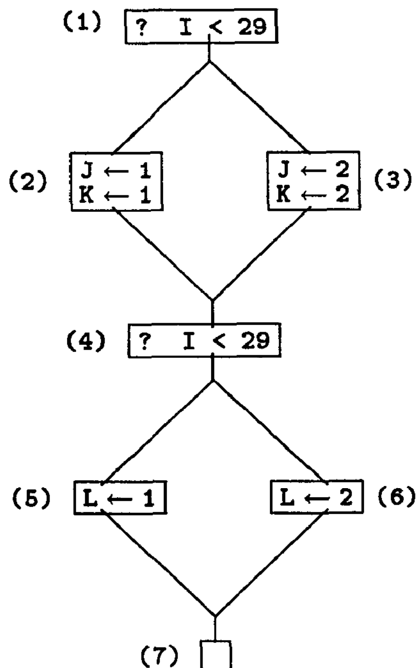

Figure 2: Control Flow Graph of Simple Example

2.2 Translation to Static Single Assignment Form

The translation to *static single assignment* (or *SSA*) form involves separating each variable V in the program into several variables V_i, each of which has only one assignment. When node i contains an assignment to V, we replace the last assignment to V, at that node, by an assignment to V_i. The uses of the variable V are replaced by the appropriate variable V_i, where i dominates the use. In order to accomplish this we need to introduce additional assignments at join points, so that there is always a dominating assignment. These *pseudo-assignments* will be of the form $X \leftarrow \phi(Y, Z)$, which means that if control reaches this node along the first entering edge, X is assigned the value Y, and if along the other edge[1] X is assigned the value of Z. These pseudo-assignments allow us to always determine at value of X, given the correct incoming edge. The SSA form for Figure 1 is given in Figure 3.

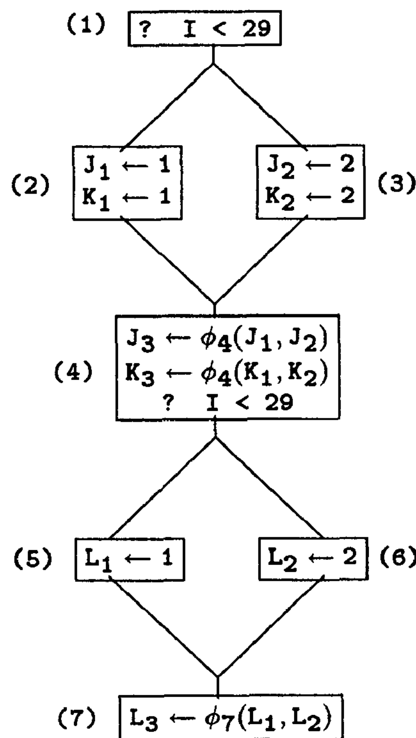

Figure 3: SSA Form of Simple Example

Following Shapiro and Saint [SS70], we describe the nodes at which ϕ's must be inserted as *pseudo-assignments*. A node n gets a pseudo-assignment for a variable X if and only if there are two paths to n from distinct assignments or pseudo-assignments for X such that n is the only node common to the two paths. A naive algorithm which runs in $O(N^2)$ for each variable can be constructed which marks all nodes which can be reached from each assignment and pseudo-assignment, stopping the marking when another assignment (or pseudo-assignment) is reached. The first node which is marked by two different assignments is another pseudo-assignment. The process is then repeated.

Better algorithms exist for determining the locations to insert the ϕ functions. Rosen, Wegman and Zadeck [RWZ88] give a relatively straightforward algorithm for reducible algorithms that works in $O(E \log E)$ (where E is the number of edges in the control flow graph) for each variable. Reif and Tarjan [RT82] give a complex algorithm that works for irreducible graphs in $O(E\alpha E)$ (where α is the inverse of Ackerman's function) bit vector operations where the number of bits grows with the number of variables[2].

For each node n which is a pseudo-assignment, we will create a *join* ϕ function ϕ_n. This function has as many

[1] If there are more than two entering edges, the ϕ will take the appropriate number of arguments.

[2] Reif and Tarjan call pseudo-assignments *join birthpoints*.

arguments as there are entering edges to the node. The value of this function is assigned to a new variable V_n, and all uses of the original variable V that are dominated by n are replaced by V_n. A variable V_n is *present* at a node a if on all paths to a the last assignment or pseudo-assignment to V took place at n. The ith argument of the ϕ whose value is assigned to V_n is the variable present at the node u where the edge (u,n) is the ith edge into to node n.

2.3 Building the Value Graph

We will build a *value graph* representing symbolic execution of the program. The value graph is a labeled, directed graph. Figure 4 is the value graph of the simple example. Each edge of the value graph corresponds to a connection between the use of a variable and the assignment at which the value of that variable is *generated*. The generating assignment for the use of a variable is normally the assignment to that variable. However, if this assignment is a *trivial assignment* (one of the form $A \leftarrow B$), then the generating assignment for A is the generating assignment for B.

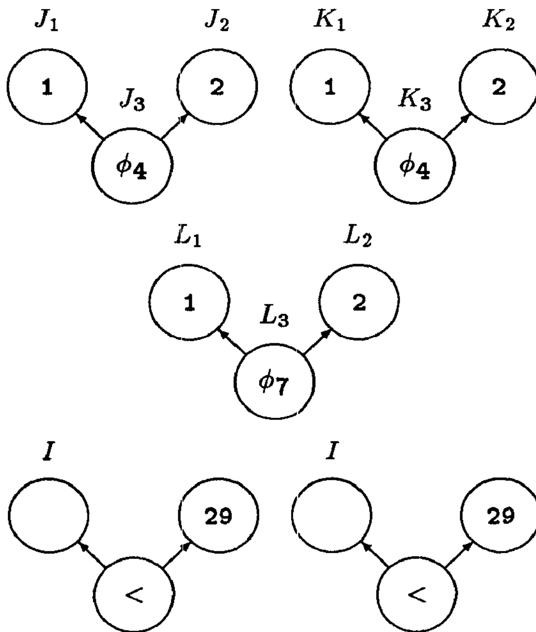

Figure 4: Value Graph of Simple Example

Each node of the value graph corresponds to an individual function in the program. The graph has two types of nodes, corresponding to the two types of assignments:

Executable Function For every normal assignment in the program, there is a node labeled with the function symbol on the right-hand side of the assignment. One edge leaves the node for each argument to the function.

ϕ Function For each pseudo-assignment there is a node labeled by the ϕ function for that join point. One edge leaves the node for each argument of the ϕ function (i.e., the nodes corresponding to the assignments that reach the join point).

The edges are ordered corresponding to the order of the arguments of the function.

Figure 4 shows the value graph for our simple example. For purposes of exposition, we have placed variable names next to the nodes that they are associated with. These names are not part of the graph. The empty node in the value structure for the predicate is a consequence of the example being incomplete. This node would normally represent the calculation producing I.

2.4 Congruence

We will show that two variables have the same value at a point p if the nodes in the value graph corresponding to the assignments to these variables are congruent and if both these assignments dominate p. Two nodes in the value graph are said to be *congruent* if both of the following conditions hold:

1. the nodes have identical function labels.
2. the corresponding destinations of the edges leaving the nodes are congruent.

As we have defined congruence, it is a symmetric, reflexive and transitive relation.

In the program fragment in Figure 4, the following are congruence classes:

- (J_1, K_1, L_1)
- (J_2, K_2, L_2)
- (J_3, K_3)
- (L_3)

Notice that L_3 is in a different congruence class from J_3 and K_3, since the two different join nodes have different ϕ functions associated with them.

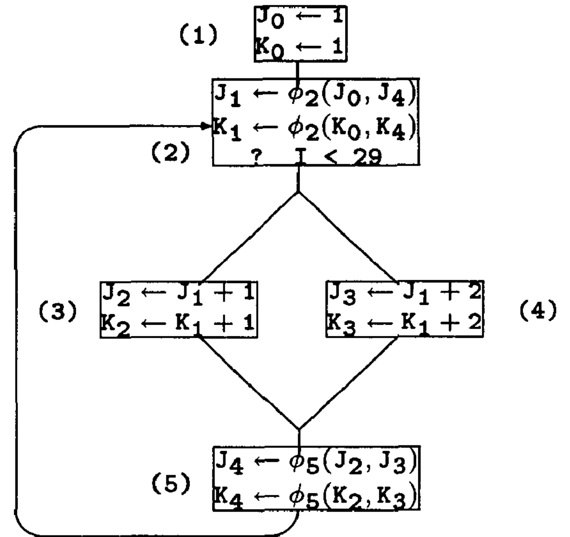

Figure 5: Control Flow Graph of Loop Example

The previous conditions are not enough to define congruence because they are circular and in fact ambiguous. The ambiguity allows multiple solutions. We desire the one that gives the maximal number of congruences. Congruence is defined as the maximal fixed point that satisfies the

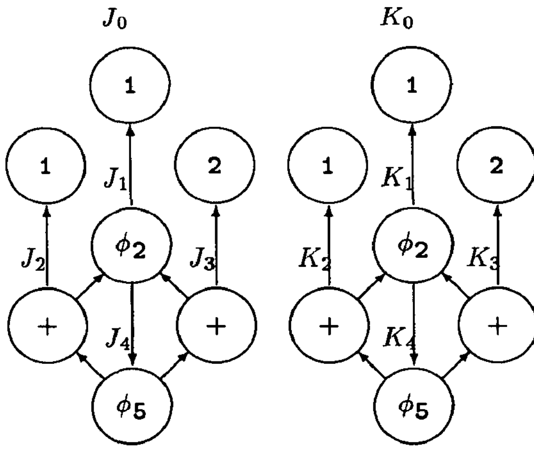

Figure 6: Value Graph of Loop Example

previous two conditions. The solution is determined *optimistically.* That is, all variables are assumed to be the same initially and this assumption is refined until a fixed point is reached. It is possible to compute a *pessimistic* solution by assuming that the variables are different and combining those that can be proven congruent. The fixed point found by a pessimistic algorithm will not, in general, be maximal, as shown by Figure 5 and 6.

While it is true that the value graphs for J and K in Figure 6 are identical, a pessimistic algorithm cannot determine that the nodes are congruent because of the cycles. By definition, for two nodes to be congruent, the destinations of all of the edges must be the same. The cycle inhibits discovering these congruences, unless they are assumed from the start.

It is not necessary to represent the congruences as pairs since the congruence relation is transitive, reflexive and commutative and moreover, there are $O(N^2)$ possible pairs. Rather, we represent them by sets of congruent nodes. The collection of all of the sets of nodes is called a *partitioning* of the nodes. Each set is called a *partition.*

2.5 Two Partitioning Algorithms

We begin with an initial partitioning of the nodes that puts all possibly congruent nodes in the same partition. (Non-congruent nodes may also start out in the same partition.) We then create a new partitioning at each step of the algorithm. Each partitioning is a refinement of the previous one. In the final partitioning, two non-congruent nodes must be in different partitions.

The following simple algorithm can be used to discover congruent nodes in the value graph.

Simple Algorithm

step 1: Place all nodes with the same label in the same partitions.

step i+1: Two nodes will be in the same partition in partitioning i+1 if, in partitioning i, the nodes are in the same partition and the corresponding destinations of their edges are in the same partitions.

The algorithm terminates when two successive partitioning are identical and takes $O(N^2)$ operations.

Aho, Hopcroft, and Ullman [AHU74] give a fast algorithm for partitioning based on an algorithm by Hopcroft [Hop71] for minimizing a finite-state machine. We will use the fast partitioning algorithm directly from [AHU74]. It is shown in Figure 7 in a generalized form similar to one they suggest. In this algorithm, the input is an initial partitioning of the set into p equivalence classes, and a collection of k functions f_i from the set to itself. Intuitively, each function f_i which maps a node u to a node v corresponds to the edge (u,v) in the value graph.

The algorithm makes use of two arrays, F^{-1} and B. The elements of the ith partition are stored in B[i]. The inverse image of element x under f_m is stored in $F^{-1}[m,x]$. The final partitioning has the property that two elements, x and y, are in the same partition only if both a) they were originally in the same partition and b) there is no function such that $f_i(x)$ is not in the same partition as $f_i(y)$.

```
WAITING ← { 1, 2, ...  p }
q ← p;
while WAITING ≠ ∅ do
   select and delete an integer i from WAITING;
   for m from 1 to k do
      INVERSE ← ∅
      for x in B[i] do
         INVERSE ← INVERSE ∪F⁻¹[m,x]
      end
      for each j such that B[j]∩ INVERSE ≠ ∅ and
            B[j] ⊈ INVERSE do
         q ← q + 1
         create a new block B[q];
         B[q] ← B[j]∩INVERSE
         B[j] ← B[j] − B[q]
         if j is in WAITING
            then add q to WAITING
            else if ||B[j]|| ≤ ||B[q]||
               then add j to WAITING
               else add q to WAITING
      end
   end
end
```

Figure 7: Hopcroft's Partitioning Algorithm

To fit congruence detection into this framework, the nodes in the value graph are initially partitioned by their label. The ith function maps a node to its ith child. The final partitioning of the algorithm leaves the nodes in the same partition if and only if the nodes are congruent.

This partitioning algorithm is based on the following ideas. At each step, partitions are split. A partition is not examined unless it needs to be split. It needs to be split if two nodes in it point to nodes in separate partitions. Splitting works by creating a new partition. Nodes are moved out of the original partition into the new one. This is in such a way that the new partition has no more than half the nodes that were in the original partition. Splitting

a partition can force other partitions to be split. Hence, new partitions are placed on a queue to be examined later. We observe that a node can be in a partition which is placed on the queue only a logarithmic number of times, since each time the partition is half the size of the previous partition of which the node was a part.

In the worst case, the congruence classes can be determined in $O(E \log E)$ time, where E is the number of edges in the value graph.

The algorithms presented in this section discovers exactly the same equivalences as that of Reif and Lewis [RL82]. Their algorithm runs in $O(E\alpha E)$ time where α is the inverse of Ackerman's function. The algorithm presented in this section has two significant advantages over the one by Reif and Lewis:

1. As formulated as a partitioning problem, our algorithm is easier to understand and implement.
2. Our algorithm can be modified to discover substantially more equivalences, as shown in Sections 3 and 4.

The key results of this paper are the extensions to our algorithm that allow us to discover more equivalences. These extensions are the subject of the rest of the paper.

3 Taking Advantage of Control Structure

In this section, we show how to detect that variables are congruent when they have been produced by identical functions within identical control structures in different parts of the program. In the previous section, we created unique ϕ functions for every join point. As a result, we have not yet been able to recognize equal values produced by the same control structure in two different places in the program. In this section, we will create new ϕ functions which reflect the semantics of common *high-level* control structures. These ϕ functions can be used in several locations in the program and thus, we can recognize that two variables are congruent even when they are defined in different locations. Such a construction is similar in spirit to the PDG of Ferrante, Ottenstein, and Warren [FOW87]. The congruence algorithm of the preceding section can now discover more equivalences.

High-level control structures can either be derived from the source program by parsing techniques or can be determined from the control flow graph by analysis techniques such as those of Baker [Bak77] and Sharir [Sha80]. We will assume in the rest of this section that the high-level control structures have been identified in the SSA graph.

In this section, we provide value structures for two of the most common control structures, `if-then-else` and a very general loop. Other control structures can be handled by similar techniques. It is not necessary for the program to be wholly constructed from a given set of high-level control structures, since any parts of the program not using our high-level control structures (e.g. loops with multiple exits) can be modeled using the ϕ functions of the previous section. The consequence of using the original ϕ functions will be merely that some equivalences will not be discovered.

It is instructive to note why we do not model this as a classical dataflow analysis problem. In a typical framework, the information about variables is represented at nodes and is propagated along the edges in a control flow graph. Here, the control flow is part of the information being propagated. Hense, it would be costly to model this as a dataflow problem.

3.1 Conditional Statements

Two variables contain the same values when they are the results of assignments in conditionals provided that 1) on the corresponding branches of the `if-then-else` they are given the same values and 2) the predicates controlling the branching are congruent. Structures can be created in the value graph that allow the partitioning algorithm to determine congruences in this case.

At each identified `if-then-else`, we use a function $\phi_{\mathtt{if}}$ to combine values coming from the different branches. This function has three arguments:

1. The variable coming from the **then** side of the `if-then-else`.
2. The variable coming from the **else** side of the `if-then-else`.
3. The predicate. Note that join ϕ nodes do not contain the predicate.

Since the $\phi_{\mathtt{if}}$ contains an edge to the predicate, congruence of two $\phi_{\mathtt{if}}$ structures implies that the corresponding `if-then-else` control structure branch in the same direction. It also implies that the values computed on those branches of the `if-then-else` are also congruent.

Consider the example in Figure 1. The new value graph, shown in Figure 8, has a $\phi_{\mathtt{if}}$ associated with $\mathtt{J}_3$, $\mathtt{K}_3$, and $\mathtt{L}_3$. The first edge out of each of these three nodes will be to a node labeled with the constant function `1`; the second edge will be to a node labeled `2`; the third edge will point to a substructure representing the predicate `I < 29`. The partitioning algorithm now detects that $\mathtt{J}_3$, $\mathtt{K}_3$, and $\mathtt{L}_3$ are congruent.

Now, if we apply our previous partitioning algorithm, the values of $\mathtt{J}_3$, $\mathtt{K}_3$, and $\mathtt{L}_3$ are congruent and dominate the point at the end of the program (node `(7)` in the flow graph in Figure 2). We therefore say that they are all equivalent at that point.

3.2 Loops

Two variables contain the same values when they are the results of assignments inside loops, provided that 1) they have the same initial values; 2) they are modified in the same way; and 3) the loops will be executed the same number of times. This section will show how to construct value graphs that allow the partitioning algorithm to detect these congruences.

The loop we consider is a control structure with a single entry, some statements, a predicate under whose control the loop is exited, and some more statements. Either set of statements may be empty. This structure can easily model either a `while` loop or a `repeat-until` loop.

For identified loops we create a $\phi_{\mathtt{enter}}$ function and a $\phi_{\mathtt{exit}}$ function for each variable modified in the loop. The

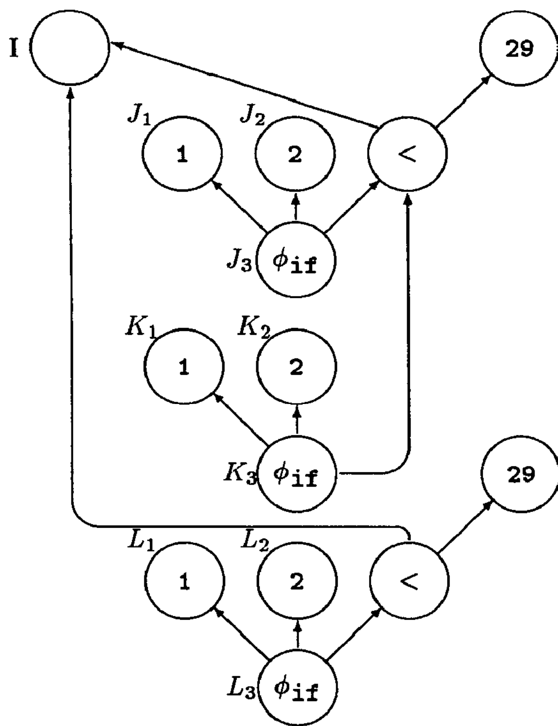

Figure 8: Better Value Graph for `if`

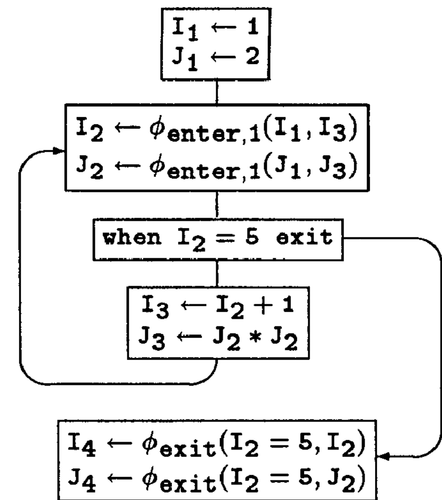

Figure 10: SSA Form of Loop Example

$\phi_{\texttt{enter}}$ functions is at the beginning of the loop. It combines the value coming into the loop and the value coming around the loop and takes two arguments:

1. The variable coming into the loop.
2. The variable that has been modified at the bottom of the loop.

The $\phi_{\texttt{exit}}$ function takes two arguments:

1. The predicate which controls when the loop is exited.
2. The variable whose assignment dominates that point.

For reasons, to be described later, there are distinct $\phi_{\texttt{enter}}$ functions for each nesting level of loops. Thus, in a program in which the deepest-nested loop is three-deep, there are three distinct $\phi_{\texttt{enter}}$ functions.

Again, the partitioning algorithm is used to determine the congruences.

```
I ← 1
J ← 2
while (I ≠ 5) do
    I ← I + 1
    J ← J * S
end
```

Figure 9: Loop Example

Consider the program in Figure 9 and it's associated SSA form in Figure 10. Both `I` and `J` are modified in the loop, and hence we have two $\phi_{\texttt{enter}}$s at the beginning of the loop. Since the loop is an outermost loop, we use the $\phi_{\texttt{enter},1}$ function. Note that the first argument to the $\phi_{\texttt{exit}}$ which assigns to J_4 depends only on I_2 and not on a value of `J`, since `J` does not participate in the predicate.

Now let us consider why we have separate $\phi_{\texttt{enter}}$ functions for each nesting level. Inside the innermost loop in Figure 11, the values of `I` and `J` may be different even though `I` and `J` have the same initial value, are incremented by the same amount, and have the same exit conditions. The reason for the possible differewnce in values is that the loops may be on different iterations. By incorporating the nesting level into the ϕ we prevent the partitioning algorithm from determining that the variables are congruent.

```
I ← 1
J ← 1
while (I ≠ 17) do
    J ← 1
    while (J ≠ 17) do
        J ← J + 1
    end
    I ← I + 1
end
```

Figure 11: Nested Loop Program

3.3 Useful Examples

In this section we present two examples to illustrate the power of our analysis.

The first example is provided by arrays. Arrays (and other structured variables) can be modeled very easily. A use of the `i`th element of an array, `A[i]`, can be modeled as the result of the function `subscript` applied to `A` and `i`. Assignments to `A[i]` can be modeled by an assignment to `A` of an array computed by the `update` function which takes the array, an index to change, and a new value. Thus,

`A[i] ← A[i] + 1`

becomes

`A ← update(A, i, subscript(A, i) + 1).`

The algorithm will detect that `A` and `B` are equivalent at the end of the program fragment in Figure 12.

```
A ← B
for I ← 1 to 10 do
    A[I] ← A[I] + I;
for J ← 1 to 10 do
    B[J] ← B[J] + J;
```

Figure 12: Pascal Array Example

```
if InListP(List, Key)
   then ChangeList(List, Key, Value)
   else InsertList(List, Key, Value)

...
function InListP(P, Key):boolean
   while (P ≠ ∅ ∧ P↑.Key) do
      P ← P↑.Next
   end
   return P↑.Key ← Key
end InListP

function ChangeList(P, Key, NewValue)
   while (P ≠ ∅ ∧  P↑.Key) do
      P ← P↑.Next
   end
   if (P↑.Key = Key)
      then P↑.Value ← NewValue
      else call Error
end ChangeList
```

Figure 13: Example of Data Abstraction

The second example can be used to compensate for a common inefficiency introduced by the use of data abstraction. One disadvantage of data abstraction is that the programmer has little control over the code generated. For example, a programmer might well write code such that shown in Figure 13.

However, if the operations **InListP** and **ChangeList** are written straight-forward manner, then **ChangeList** will have to traverse the list that was already traversed by **InListP**. Assuming that the code has been expanded inline, however our algorithm will discover that the two loop traversals are the same. Other algorithms may then be able to exploit this fact by combining the loops.

Pointers are handled in the same way as arrays. In fact, every subfield can be modeled as an array. Thus, in the above example, there would be an separate array for **Key, Value** and **Next**.

3.4 The Fundamental Theorem and Its Proof

In this section, we prove that two variables are equivalent at a point **p** if they are congruent and if their defining assignments dominate **p**. We prove this as the corollary to a more general theorem dealing with a dynamic notion that implies the static notion of dominance. To prove this theorem, we first prove a lemma that allows us to replace the loops in a program with functions. Before presenting the lemma, however we need to define some terms.

The *scope* of an **if-then-else** is all the statements, including the pseudo-assignments at its end. The *scope* of a **loop** is all the statements, including the pseudo-assignments at the end, contained within it.

A node **n** is an *input* to a executable function node **m** of the value graph if there is an edge from **m** to **n**. A node **n** is an input to a $\phi_{\mathbf{if}}$ or $\phi_{\mathbf{exit}}$ node **m** if there is a path from **m** to **n** in the value structure and if **n** is the first node not contained in the scope of **m**.

A *well-formed substructure* of a value structure contains the root of the value structure and a set of nodes. All the nodes contained in the structure are reachable from the root. If a node **n** is in a well-formed substructure, then all nodes on the path from **n** to its inputs are in the substructure. The inputs of the substructure are all nodes **m** not in the substructure but such that there is an edge to **m** from a node in the substructure.

A well-formed substructure node is *pinned* if it contains a join ϕ.

We construct a function for a well-formed and unpinned substructure as follows. If the root of the structure is an executable function, recursively construct the functions for its arguments and append the application of the executable function to the evaluation of its arguments. If the root of the structure is a $\phi_{\mathbf{if}}$, then construct a function that evaluates the recursively created function for the test child of the root and then evokes the recursively created function for the appropriate branch. If the root of the structure is a $\phi_{\mathbf{exit}}$, then construct the functions for the inputs to the loop, and construct the functions for the well-formed substructures consisting of the second child of each of the $\phi_{\mathbf{enter}}$ nodes (i.e. the values coming around the loop). Construct the function which first evaluates the inputs to the loop and stores them. The function then repeatedly evaluates the test (i.e. the first child of the $\phi_{\mathbf{exit}}$) and if the test fails, it evaluates the functions for the second argument to the $\phi_{\mathbf{enter}}$ functions applied to the stored values, and stores the result. When the test succeeds, the result of the value function (i.e. the second child of the $\phi_{\mathbf{exit}}$) is returned.

Note that these functions are constructed in such a way that they perform the same calculation on their inputs as the original program.

Lemma: If two nodes are congruent and the smallest well-formed substructures containing them are unpinned, then the functions derived from them are identical.

The proof follows from an induction on the size of the smallest well-formed substructures.

Let a variable be *active* at a moment during execution if and only if either of the following two conditions is satisfied:

1. The defining assignment of the variable is not contained in a loop and has already been executed.
2. The innermost loop containing the defining assignment of the variable is currently executing and the assign-

ment has already been executed during the current iteration.

Theorem: Two active, congruent variables have the same value.

Proof of the Theorem: The proof proceeds by contradiction. Execution is stopped at the first moment at which the theorem breaks down. Assume **x** and **y** are active, congruent variables that have different values. Without loss of generality, assume also that the defining assignment of **x** has just been executed.

Let **r** and **s** be the roots of the value structures for **x** and **y**, respectively. We will derive a contradiction for each possible labeling of **r**.

Case 1: **r** is labeled by either a function symbol **f**, or an unpinned $\phi_{\mathtt{if}}$, or an unpinned $\phi_{\mathtt{exit}}$.

By the lemma, the values of **x** and **y** are computed by the same function of their inputs, since they are congruent. Furthermore, because they are congruent, their inputs must be congruent. Since **x** and **y** have different values, at least one pair of their inputs, **x**′ and **y**′, must have different values. By construction, the inputs to a structure are active at the beginning of the structure. Therefore **x**′ and **y**′ are active at the beginning of the scopes that define **x** and **y**, respectively. Also, anything active at the beginning of a scope is active at the end (by definition). Therefore, **x**′ and **y**′ are active at the definitions of **x** and **y**, respectively. Finally, if **a** is active at the definition of **b** then **a** is active wherever **b** is. Therefore, **y**′ is active at the definition of **x** (since **y** is). Thus, **x**′ and **y**′ are active, congruent variables with different values at the definition of **x**. This contradicts our assumption that the theorem first breaks down after the definition of **x**.

Case 2: **r** is labeled by a join ϕ, or a pinned $\phi_{\mathtt{if}}$ or a pinned $\phi_{\mathtt{exit}}$. The defining assignments to **x** and **y** occur at the same node, since otherwise the value structures would not be congruent. These assignments give **x** and **y** values of variables that were active immediately preceding the node. The value structures for these variables are corresponding substructures of the value structures for **x** and **y**, and hence are congruent. The contradiction follows immediately.

Case 3: **r** is labeled $\phi_{\mathtt{enter}}$. Since both **x** and **y** are active, the innermost loop containing the definition of **y** must contain the innermost loop containing the definition of **x**. Since **x** and **y** are congruent, the loops for **x** and **y** must be nested at the same depth, since there are distinct $\phi_{\mathtt{enter}}$ functions for each level of nesting. Thus, **x** and **y** are defined in the same loop. If **x** and **y** have different values before the first iteration of the loop, then the congruent variables corresponding to the first children of the roots of the value structures for **x** and **y** have different values before the loop. If **x** and **y** have different values after the *n*th iteration of the loop, then the congruent variables corresponding to their second children have different values before the end of the *n*th iteration. In either case there is a contradiction.

QED

Corollary: Two congruent variables are equivalent at a point **p** if their definitions dominate **p**.

The corollary follows immediately from the observation that a variable must be active at a point if its defining assignment dominates the point.

4 Simple Modifications to the Partitioning Algorithm

Code optimization is a combination of analysis and transformation. These can feed on each other synergistically. Performing a transformation may provide an opportunity that can be exposed by further analysis. This can permit a second transformation that may expose further opportunities. As a result, there has been considerable discussion of the proper ordering of optimizations [Pol86] and several authors have suggested repeated invocations of one or more optimizations to achieve a better result [AH82] [RWZ88].

Some optimizations can be combined into a single algorithm which can do more than can repeated invocations of the separate transformations [Weg75] [WZ85]. The advantage of combining over repeated invocation occurs when each analysis can be performed optimistically. An optimistic analysis proceeds by making too many assumptions and then eliminating any of those assumptions that it cannot justify on the basis of the state of the current analysis. If two methods of analysis are running in parallel, it may be that some assumptions need not be dropped because the other analysis method can justify it.

The algorithm presented here can be profitably combined with other optimizations. This can be done either by iteratively repeating them or by unifying them. As discussed above, unification may produce a more powerful algorithm which may also be considerably more complex. The decision to unify must be based on engineering issues.

There are many possible transformations that could be combined. In this section, we will discuss three such transformations and will show how to integrate the last two transformations into this algorithm.

4.1 Constant Propagation

The algorithm can benefit from other kinds of preprocessing. Function symbols need not be entirely uninterpreted. Reif and Lewis [RL77] perform constant propagation before their algorithm, a strategy which has several advantages.

Our congruence algorithm deals with variables in a purely symbolic manner and does not interpret functions. It is easy to interpret the behavior of functions on constant operands at compile time; one simply applies them and determines the result.

While it is true that the algorithm presented here may provide further opportunities to discover constants, unification of the two algorithms will be complex.

4.2 Symmetry

Our requirement that functions be uninterpreted might be relaxed in order to allow the observation that addition is commutative. Thus, we know that **33+17** equals **17+33**

(even if we don't know that both equal 50). The partition algorithm would not discover this equivalence since the first edge out of the first node points to a node labeled 33, while the first edge out of the second points to a node labeled 17. If we knew which equivalence classes the children of the + were in at preprocessing time, then they could be sorted. But we don't. The solution is to allow the + node to have a single *hyperedge* to both of its children.

This solution will require that the partitioning algorithm handle functions whose range is sets of elements rather than just single elements. The function from the + node has as its value the set containing both of its children. This requires no change in how the inverse[3] of a function is stored. However, it will no longer be sufficient to split a partition (e.g. B[j]) into two pieces (e.g. B[j] and B[q]) depending on whether or not an element is in the inverse image of the splitting partition (e.g. B[i]). Rather, a partition will be split into several pieces depending on the number of times an element occurs as an inverse of the other partition.

The modifications to the algorithm are: 1) partition B[j] may have to be split even if it is wholly contained in INVERSE; 2) when a partition is split, two elements will end up in different partitions if they have a different number of children in B[i]; and 3) but the largest partition is placed on the WAITING queue (as before, when there was a single non-largest partition to put on the queue).

4.3 Combining Congruent Values

Suppose we wish to recognize that X and Y are equal after the following program segment:

```
X ← A+B
if P
    then Y ← A+B
    else Y ← X
```

Then we need to extend our definition of congruence. Two nodes are *congruent* if and only if they have the same label and their children are congruent (as before), or if one of the nodes is labeled $\phi_{\texttt{if}}$ and its first two children are congruent to the other.

Our strategy will be to transform the value graph as if we knew that the **then** branch and the **else** branch of an **if-then-else** produced the same value. As the algorithm proceeds, it may discover that this assumption was false. It will then undo the transformation.

The transformation is to replace all edges to a $\phi_{\texttt{if}}$ with edges to its **then** branch (this has the same effect as using the **else** branch, since we are assuming they are equal). This transformation can be applied in linear time by proceeding upwards from the leaves of the value structure. We also optimistically delete the $\phi_{\texttt{if}}$ nodes from the graph (and from any partitioning thereof). In an auxiliary data structure, we store the pairs of nodes that are optimistically assumed equal and the $\phi_{\texttt{if}}$ which had them as its children.

When a partition is split, the nodes that are split from the partition (e.g. B[q]) are examined. If one of these elements has a partner that was assumed to be equal to it, then that partner is further examined. If it is in the same partition (e.g. B[q]) or not in the value graph, we ignore it. If it is in another partition, then we undo the transformation. This will cause a $\phi_{\texttt{if}}$ to be placed into the value graph and will reattach edges to it. (If this new node belongs in a partition with other elements they will be created at the same step.) The new nodes in the value graph must be examined and tested to see if they have partners. If so, the process is repeated.

This procedure can still be implemented in $O(E \log E)$.

5 Conclusion

The detection of equivalence of variables is an undecidable problem. We have presented an algorithm which detects many of the statically detectable classes of equalities. Our algorithm is efficient and will, we hope, be fairly easy to understand and implement.

It should be possible to extend the techniques presented here so as to detect additional classes of equalities. An engineering decision will have to be made about which additional improvements to the algorithm are worth their implementation expense. As programming practice changes and as different optimizations are added to programming languages, these decisions may change.

6 Acknowledgements

We would like to thank Len Berman, Larry Carter, Ron Cytron, Jean Ferrante, Brent Hailpern, Susan Horwitz and Barry Rosen for their help in this work.

[3] We are stretching notation here. Technically, the inverse of the function takes a set of elements as its argument. We say that the inverse of the function applied to an element yields those elements the function maps to sets containing the element.

References

[AH82] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, 17(6):22–31, June 1982.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[All70] F. E. Allen. Control flow analysis. *Sigplan Notices*, July 1970.

[Bak77] B. S. Baker. An algorithm for structuring flowgraphs. *J. ACM*, 98–120, January 1977.

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

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. A program dependance graph and it's use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GW76] S. L. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, January 1976.

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

[Pol86] L. L. Pollock. *An Approach to Incremental Compilation of Optimized Code.* PhD thesis, Department of Computer Science, University of Pittsburgh, Pittsburgh, Pa. 15260, 1986.

[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*, 104–118, January 1977.

[RL82] J. H. Reif and H. R. Lewis. *Efficent Symbolic Analysis of Programs.* Technical Report TR-37-82, Harvard University, Aiken Computation Laboratory, 33 Oxford St.,Cambridge, Mass 02138, 1982.

[RT82] J. H. Reif and R. E. Tarjan. Symbolic program analysis in almost linear time. *SIAM J. Computing*, 11(1):81–93, February 1982.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, January 1988.

[Sha80] M. Sharir. Structural analysis: a new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.

[SS70] R. M. Shapiro and H. Saint. *The Representation of Algorithms.* Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.

[Weg75] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. on Software Engineering*, SE-1(3):270–285, September 1975.

[WZ85] M. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *Conf. Rec. Twelfth ACM Symp. on Principles of Programming Languages*, 291–299, January 1985.