Published in "Static Analysis Symposium" (SAS), 2004, pages 212-227

A Polynomial-Time Algorithm for Global Value Numbering

Sumit Gulwani and George C. Necula

University of California, Berkeley
{gulwani,necula}@cs.berkeley.edu

Abstract. We describe a polynomial-time algorithm for global value numbering, which is the problem of discovering equivalences among program sub-expressions. We treat all conditionals as non-deterministic and all program operators as uninterpreted. We show that there are programs for which the set of all equivalences contains terms whose value graph representation requires exponential size. Our algorithm discovers all equivalences among terms of size at most s in time that grows linearly with s. For global value numbering, it suffices to choose s to be the size of the program. Earlier deterministic algorithms for the same problem are either incomplete or take exponential time.

1 Introduction

Detecting equivalence of program sub-expressions has a variety of applications. Compilers use this information to perform several important optimizations like constant and copy propagation [13], common sub-expression elimination, invariant code motion [2,11], induction variable elimination, branch elimination, branch fusion, and loop jamming [8]. Program verification tools use these equivalences to discover loop invariants, and to verify program assertions. This information is also important for discovering equivalent computations in different programs; this is useful for plagiarism detection tools and translation validation tools [10,9], which compare a program with an optimized version in order to check the correctness of the optimizer.

Checking equivalence of program expressions is an undecidable problem, even when all conditionals are treated as non-deterministic. Most tools, including compilers, attempt to only discover equivalences between expressions that are computed using the same operator applied to equivalent operands. This form of equivalence, where the operators are treated as uninterpreted functions, is also called *Herbrand equivalence* [12]. The process of discovering such restricted class of equivalences is often referred to as *value numbering*. Performing value numbering in basic blocks is an easy problem; the challenge is in doing it globally for a procedure body.

Existing deterministic algorithms for global value numbering are either too expensive or imprecise. The precise algorithms are based on an early algorithm by Kildall [7], which discovers equivalences by performing an abstract interpretation [3] over the lattice of Herbrand equivalences. Kildall's algorithm discovers all

Herbrand equivalences in a function body but has exponential cost [12]. On the other extreme, there are several polynomial-time algorithms that are complete for basic blocks, but are imprecise in the presence of joins and loops in a program. The popular partition refinement algorithm proposed by Alpern, Wegman, and Zadeck (AWZ) [1] is particularly efficient, however at the price of being significantly less precise than Kildall's algorithm. The novel idea in AWZ algorithm is to represent the values of variables after a join using a fresh selection function ϕ_i, similar to the functions used in the static single assignment form [4], and to treat the ϕ_i functions as additional uninterpreted functions. The AWZ algorithm is incomplete because it treats ϕ functions as uninterpreted. In an attempt to remedy this problem, Rüthing, Knoop and Steffen have proposed a polynomial-time algorithm (RKS) [12] that alternately applies the AWZ algorithm and some rewrite rules for normalization of terms involving ϕ functions, until the congruence classes reach a fixed point. Their algorithm discovers more equivalences than the AWZ algorithm, but remains incomplete. The AWZ and the RKS algorithm both use a data structure called value graph [8], which encodes the abstract syntax of program sub-expressions, and represents equivalences by merging nodes that have been discovered to be referring to equivalent expressions. We discuss these algorithms in more detail in Section 5. Recently, Gargi has proposed a set of balanced algorithms that are efficient, but also incomplete [5].

Our algorithm is based on two novel observations. First, it is important to make a distinction between "discovering all Herbrand equivalences" vs. "discovering Herbrand equivalences among program sub-expressions". The former involves discovering Herbrand equivalences among all terms that can be constructed using program variables and uninterpreted functions in the program. The latter refers to only those terms that occur syntactically in the program. Finding all Herbrand equivalences is attractive not only to answer questions about non-program terms, but it also allows a forwards dataflow or abstract interpretation based algorithms (e.g. Kildall's algorithm) to discover all equivalences among program terms. This is because discovery of an equivalence between program terms at some program point may require detecting equivalences among non-program terms at a preceding program point. This distinction is important because we show (in Section 4) that there is a family of acyclic programs for which the set of all Herbrand equivalences requires an exponential sized (in the size of the program) value graph representation. On the other hand, we also show that Herbrand equivalences among program sub-expressions can always be represented using a linear sized value graph. This implies that no algorithm that uses value graphs to represent equivalences can discover all Herbrand equivalences and have polynomial-time complexity at the same time. This observation explains why existing polynomial-time algorithms for value numbering are incomplete, even for acyclic programs. One of the reasons why Kildall's algorithm is exponential is that it discovers all Herbrand equivalences at each program point.

The above observation not only sheds light on the incompleteness or exponential complexity of the existing algorithms, but also motivates the design of

our algorithm. Our algorithm takes a parameter s and discovers all Herbrand equivalences among terms of size at most s in time that grows linearly with s. For the purpose of global value numbering, it is sufficient to set the parameter s to N, where N is the size of the program, since the size of any program expression is at most N.

The second observation is that the lattice of sets of Herbrand equivalences has finite height k, where k is the number of program variables (we prove this in Section 3.4). Therefore, an optimistic-style algorithm that performs an abstract interpretation over the lattice of Herbrand equivalences will be able to handle cyclic programs as precisely as it can handle acyclic programs, and will terminate in at most k iterations. Without this observation, one can ensure the termination of the algorithm in presence of loops by adding a degree of pessimism. This leads to incompleteness in presence of loops, as is the case with the RKS algorithm [12]. Instead, our algorithm is based on abstract interpretation, similar to Kildall's algorithm, while using a more sophisticated join operation. We continue with a description of the expression language on which the algorithm operates (in Section 2), followed by a description of the algorithm itself in Section 3.

2 Language of Program Expressions

We consider a language in which the expressions occurring in assignments belong to the following simple language of uninterpreted function terms (here x is one of the variables, and c is one of the constants):

$$e ::= x \quad | \quad c \quad | \quad F(e_1, e_2)$$

For any expression e, we use the notation $\mathit{Variables}(e)$ to denote the variables that occur in expression e. We use $\mathit{size}(e)$ to denote the number of occurrences of function symbols in expression e (when expressed as a value graph). For simplicity, we consider only one binary uninterpreted function F. Our results can be extended easily to languages with any finite number of uninterpreted functions of any constant arity. Alternatively, one can encode any finite number of uninterpreted functions of constant arity by *one* binary function symbol with only a constant factor increase in the size of the program.

3 The Global Value Numbering Algorithm

Our algorithm discovers the set of Herbrand equivalences at any program point by performing an abstract interpretation over the lattice of Herbrand equivalences. We pointed out in the introduction, and we argue further in Section 4, that we cannot hope to have a complete and polynomial-time algorithm that discovers all Herbrand equivalences implied by a program (using the standard value graph based representations) because their representation is worst-case exponential in the size of the program. Thus, our algorithm takes a parameter s (which is a positive integer) and discovers all equivalences of the form $e_1 = e_2$,

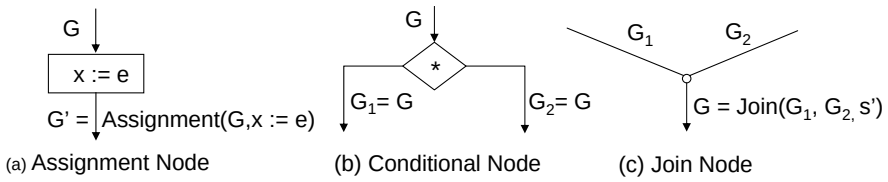

Fig. 1. Flowchart nodes

where $size(e_1) \leq s$ and $size(e_2) \leq s$. The algorithm uses a data structure called *Strong Equivalence DAG* (described in Section 3.1) to represent the set of equivalences at any program point. It updates the data structure across each flowchart node as shown in Figure 1. The `Assignment` and `Join` functions are described in Section 3.2 and Section 3.3 respectively.

3.1 Notation and Data Structure

Let T be the set of all program variables, k the total number of program variables, and N the size of the program, measured in terms of the number of occurrences of function symbol F in the program.

The algorithm represents the set of equivalences at any program point by a data structure that we call *Strong Equivalence DAG* (SED). An SED is similar to a value graph. It is a labeled directed acyclic graph whose nodes can be represented by tuples $\langle V, t\rangle$ where V is a (possibly empty) set of program variables labeling the node, and t represents the type of node. The type t is either $\perp$ or c, indicating that the node has no successors, or $F(n_1, n_2)$ indicating that the node has two ordered successors n_1 and n_2.

In any SED G, for every variable x, there is exactly one node $\langle V, t\rangle$, denoted by $Node_G(x)$, such that $x \in V$. For every type t that is not $\perp$, there is at most one node with that type. We use the notation $Node_G(c)$ to refer to the node with type c. For any SED node n, we use the notation $Vars(n)$ to denote the set of variables labeling node n, and $Type(n)$ to denote the type of node n. Every node n in an SED represents the following set of terms $Terms(n)$, which are all known to be equivalent.

$$\begin{aligned} Terms(V, \perp) &= V \\ Terms(V, c) &= V \cup \{c\} \\ Terms(V, F(n_1, n_2)) &= V \cup \{F(e_1, e_2) \mid e_1 \in Terms(n_1), e_2 \in Terms(n_2)\} \end{aligned}$$

We use the notation $G \models e_1 = e_2$ to denote that G implies the equivalence $e_1 = e_2$. The judgment $G \models e_1 = e_2$ is deduced as follows.

$$\begin{aligned} G \models F(e_1, e_2) = F(e_1', e_2') \text{ iff } G \models e_1 = e_1' \text{ and } G \models e_2 = e_2' \\ G \models x = e \text{ iff } e \in Terms(Node_G(x)) \end{aligned}$$

The algorithm starts with the following initial SED at the program start, which implies only trivial equivalences.

$$G_0 = \{\langle x, \perp\rangle \mid x \in T\}$$

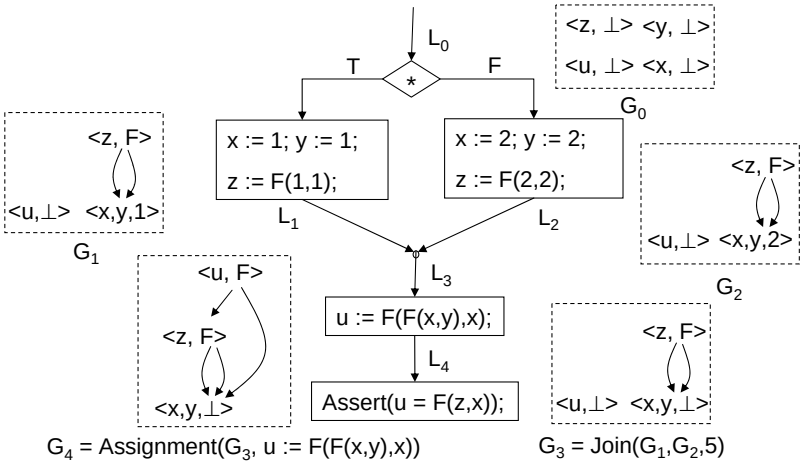

Fig. 2. This figure shows a program and the execution of our algorithm on it. G_i, shown in dotted box, represents the SED at program point L_i.

In figures showing SEDs, we omit the set delimiters "{" and "}", and represent a node $\langle\{x_1, .., x_n\}, t\rangle$ as $\langle x_1, .., x_n, t\rangle$. Figure 2 shows a program and the SEDs computed by our algorithm at various points. As an example, note that $Terms(Node_{G_4}(u)) = \{u\} \cup \{F(z, \alpha) \mid \alpha \in \{x, y\}\} \cup \{F(F(\alpha_1, \alpha_2), \alpha_3) \mid \alpha_1, \alpha_2, \alpha_3 \in \{x, y\}\}$. Hence, $G_4 \models u = F(z, x)$. Note that an SED represents compactly a possibly-exponential number of equivalent terms.

3.2 The Assignment Operation

Let G be an SED that represents the Herbrand equivalences before an assignment node $x := e$. The SED that represents the Herbrand equivalences after the assignment node can be obtained by using the following algorithm. SED G_4 in Figure 2 shows an example of the `Assignment` operation.

```
1 Assignment(G, x := e) =
2   G' := G;
3   let ⟨V1, t1⟩ = GetNode(G', e) in
4   let ⟨V2, t2⟩ = Node_G'(x) in
5   if t1 ≠ t2 then G' := G' − {⟨V1, t1⟩, ⟨V2, t2⟩};
6                   G' := G' ∪ {⟨V1 ∪ {x}, t1⟩, ⟨V2 − {x}, t2⟩};
7   return G';

1 GetNode(G', e) =
2   match e with
3     y: return Node_G'(y);
4     c: return Node_G'(c);
5     F(e1, e2): let n1 = GetNode(G', e1) and n2 = GetNode(G', e2) in
6                if ⟨V, F(n1, n2)⟩ ∈ G' for some V, then return ⟨V, F(n1, n2)⟩;
7                else G' := G' ∪ ⟨∅, F(n1, n2)⟩; return ⟨∅, F(n1, n2)⟩;
```

`GetNode`(G', e) returns a node n such that $e \in Terms(n)$ (and in the process possibly extends G') in $O(size(e))$ time. Lines 5 and 6 in `Assignment` function move variable x to node n to reflect the new equivalence $x = e$. Hence, the following lemma holds.

Lemma 1 (Soundness and Completeness of Assignment Operation). *Let* $G' =$ `Assignment`$(G, x := e)$. *Let* e_1 *and* e_2 *be two expressions. Let* $e_1' = e_1[^{e}/_{x}]$ *and* $e_2' = e_2[^{e}/_{x}]$. *Then,* $G' \models e_1 = e_2$ *iff* $G \models e_1' = e_2'$.

3.3 The Join Operation

Let G_1 and G_2 be two SEDs. Let s' be any positive integer. The following function `Join` returns an SED G that represents all equivalences $e_1 = e_2$ such that both G_1 and G_2 imply $e_1 = e_2$ and both $size(e_1)$ and $size(e_2)$ are at most s'. In order to discover all equivalences among expressions of size at most s in the program, we need to choose $s' = s + N \times k$ (for reasons explained later in Section 3.5). Figure 2 shows an example of the `Join` operation.

For any SED G, let $\prec_G$ denote a partial order on program variables such that $x \prec_G y$ if y depends on x, or more precisely, if $G \models y = F(e_1, e_2)$ such that $x \in Variables(F(e_1, e_2))$.

```
1  Join(G1,G2,s') =
2    for all nodes n1 ∈ G1 and n2 ∈ G2, memoize[n1,n2] := undefined;
3    G := ∅;
4    for each variable x ∈ T in the order ≺_G1 do
5        counter := s';
6        Intersect(Node_G1(x), Node_G2(x));
7    return G;

1  Intersect(⟨V1,t1⟩,⟨V2,t2⟩) =
2    let m = memoize(⟨V1,t1⟩,⟨V2,t2⟩) in
3    if m ≠ undefined then return m;
4    let t = if counter > 0 and t1 ≡ F(ℓ1,r1) and t2 ≡ F(ℓ2,r2) then
5                counter := counter − 1;
6                let ℓ = Intersect(ℓ1,ℓ2) in
7                let r = Intersect(r1,r2) in
8                if (ℓ ≠ ⟨φ,⊥⟩) and (r ≠ ⟨φ,⊥⟩) then F(ℓ,r) else ⊥
9            else if t1 = c and t2 = c for some c, then c
10           else ⊥ in
11   let V = V1 ∩ V2 in
12   if V ≠ ∅ or t ≠ ⊥ then G := G ∪ {⟨V,t⟩}
13   memoize[⟨V1,t1⟩,⟨V2,t2⟩] := ⟨V,t⟩;
14   return ⟨V,t⟩
```

It is important for correctness of the algorithm that calls to the `Intersect` function are memoized, as done explicitly in the above pseudo code, since otherwise the counter variable will be decremented incorrectly. The use of counter

variable ensures that the call to `Intersect` function in `Join` terminates in $O(s')$ time. The following proposition describes the property of `Intersect` function that is required to prove the correctness of the `Join` function (Lemma 2).

Proposition 1. *Let $n_1 = \langle V_1, t_1 \rangle$ and $n_2 = \langle V_2, t_2 \rangle$ be any nodes in SEDs G_1 and G_2 respectively. Let $n = \langle V, t \rangle = \texttt{Intersect}(n_1, n_2)$. Suppose that $n \neq \langle \emptyset, \bot \rangle$; hence the function $\texttt{Intersect}(n_1, n_2)$ adds the node n to G. Let α be the value of the counter variable when $\texttt{Intersect}(n_1, n_2)$ is first called. Then,*

P1. $Terms(n) \subseteq Terms(n_1) \cap Terms(n_2)$.
P2. $Terms(n) \supseteq \{e \mid e \in Terms(n_1), e \in Terms(n_2), size(e) \leq \alpha\}$.

The proof of Proposition 1 is by induction on sum of height of nodes n_1 and n_2 in G_1 and G_2 respectively. Claim P1 is easy since $t = F(...)$ or c only if both t_1 and t_2 are $F(...)$ or c respectively (Lines 8 and 9), and $V = V_1 \cap V_2$ (Line 11). The proof of claim P2 relies on bottom-up processing of one of the SEDs, and memoization. Let e' be one of the *smallest* expressions (in terms of *size*) such that $e' \in Terms(n_1) \cap Terms(n_2)$. If e' is not a variable, then for any variable $y \in Variables(e')$, the call $\texttt{Intersect}(Node_{G_1}(y), Node_{G_2}(y))$ has already finished. The crucial observation now is that if $size(e') \leq \alpha$, then the set of recursive calls to `Intersect` are in 1-1 correspondence with the nodes of expression e', and $e' \in Terms(n)$.

Lemma 2 (Soundness and Completeness of Join Operation). *Let $G = \texttt{Join}(G_1, G_2, s)$. If $G \models e_1 = e_2$, then $G_1 \models e_1 = e_2$ and $G_2 \models e_1 = e_2$. If $G_1 \models e_1 = e_2$ and $G_2 \models e_1 = e_2$ such that $size(e_1) \leq s$ and $size(e_2) \leq s$, then $G \models e_1 = e_2$.*

The proof of Lemma 2 follows from Proposition 1 and definition of $\models$.

3.4 Fixed Point Computation

The algorithm goes around loops in a program until a fixed point is reached. The following theorem implies that the algorithm needs to execute each flowchart node at most k times (assuming the standard worklist implementation [8]).

Theorem 1 (Fixed Point Theorem). *The lattice of sets of Herbrand equivalences (involving program variables) ordered by set inclusion has height at most k where k is the number of program variables.*

The proof of Theorem 1 follows easily from Lemma 3 stated and proved below. Before stating Lemma 3, we first introduce some notation. Let $\preccurlyeq$ denote any total ordering on all program variables. For notational convenience, we say that for any variable x, and any expressions e_1 and e_2, $x \preccurlyeq F(e_1, e_2)$. For any SED G, let I_G be the set of variables x such that $Type(Node_G(x)) = \bot$, and $x \preccurlyeq y$ for all $y \in Vars(Node_G(x))$. I_G is a *maximal* set of independent variables, which occur at the leaves of G. In other words, equivalences denoted by an SED G can be represented by a set of equivalences $x = e$, where $Variables(e) \subseteq I_G$ and

$x \notin I_G$. This is because for any SED G, all equivalences $e_1 = e_2$ are consequences of equivalences of the form $x = e$. For example, consider the program in Figure 2. If $u \preccurlyeq x \preccurlyeq y \preccurlyeq z$, then $I_{G_4} = \{x\}$. Note that equivalences represented by G_4 are equivalent to the set of equivalences $\{y = x, z = F(x, x), u = F(F(x, x), x)\}$.

Lemma 3. *Let G_1 and G_2 be two SEDs. If G_2 is above G_1 in the lattice (which is to say that G_1 represents a stronger set of equivalences than G_2), then $I_{G_2} \supset I_{G_1}$.*

Proof. We first make two useful observations. Let G be any SED. Then, (a) $G \not\models x = e$ such that $x \in I_G$, $e \preccurlyeq x$ and $e \not\equiv x$. (b) $G \not\models e_1 = e_2$ such that $\mathit{Variables}(e_1) \subseteq I_G$, $\mathit{Variables}(e_2) \subseteq I_G$ and $e_1 \not\equiv e_2$.

We first show that $I_{G_2} \supseteq I_{G_1}$. Suppose for the purpose of contradiction that $I_{G_2} \not\supseteq I_{G_1}$. Then, $G_2 \models x = e$ for some variable $x \in I_{G_1}$ and expression e such that $e \preccurlyeq x$ and $e \not\equiv x$. Since G_1 represents a stronger set of equivalences, $G_1 \models x = e$. But this is not possible because of observation (a) above.

We now show that $I_{G_2} \supset I_{G_1}$. Suppose for the purpose of contradiction that $I_{G_2} = I_{G_1}$. Since G_1 is stronger than G_2, $G_1 \models x = e_1$ for some $x \in T - I_{G_1}$ and expression e_1 such that $\mathit{Variables}(e_1) \subseteq I_{G_1}$ and $G_2 \not\models x = e_1$. Note that $x \in T - I_{G_2}$ since $I_{G_2} = I_{G_1}$. Hence, there exists an expression e_2 such that $G_2 \models x = e_2$, where $\mathit{Variables}(e_2) \subseteq I_{G_2}$. Note that $e_1 \not\equiv e_2$ since $G_2 \not\models x = e_1$ and $G_2 \models x = e_2$, Since G_1 is stronger than G_2, $G_1 \models x = e_2$ and hence $G_1 \models e_1 = e_2$. But this is not possible because of observation (b) above.

3.5 Correctness of the Algorithm

The correctness of the algorithm follows from Theorem 2 and Theorem 3.

Theorem 2 (Soundness Theorem). *Let G be the SED computed by the algorithm at some program point P after fixed point computation. If $G \models e_1 = e_2$, then $e_1 = e_2$ holds at program point P.*

The proof of Theorem 2 follows directly from the soundness of the assignment operation (Lemma 1 in Section 3.2) and the soundness of the join operation (Lemma 2 in Section 3.3).

Theorem 3 (Completeness Theorem). *Let $e_1 = e_2$ be an equivalence that holds at a program point P such that $\mathit{size}(e_1) \leq s$ and $\mathit{size}(e_2) \leq s$. Let G be the SED computed by the algorithm at program point P after fixed point computation. Then, $G \models e_1 = e_2$.*

The proof of Theorem 3 follows from an invariant maintained by the algorithm at each program point. For purpose of describing this invariant, we hypothetically extend the algorithm to maintain a set S of paths at each program point (representing the set of all paths analyzed by the algorithm), and a variable MaxSize (representing the size of the largest expression computed by the program along any path in S) besides an SED. These are updated as shown in Figure 3. The initial value of MaxSize is chosen to be 0. The initial set of paths is chosen to be the singleton set containing an empty path. The algorithm maintains the following invariant at each program point.

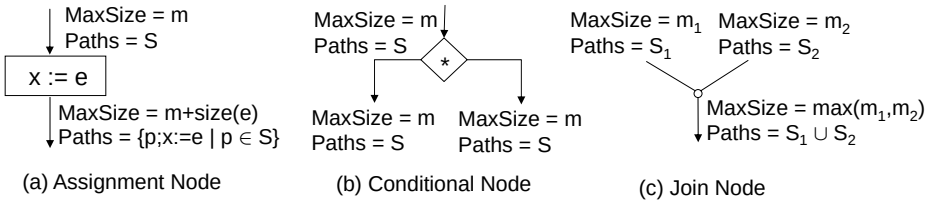

Fig. 3. Flowchart nodes

Lemma 4. *Let G be the SED, m be the value of variable MaxSize, and S be the set of paths computed by the algorithm at some program point P. Suppose $e_1 = e_2$ holds at program point P along all paths in S, $size(e_1) \leq s' - m$ and $size(e_2) \leq s' - m$. Then, $G \models e_1 = e_2$.*

Lemma 4 can be easily proved by induction on the number of operations performed by the algorithm.

Theorem 1 (the fixed point theorem) requires the algorithm to execute each node at most k times. This implies that the value of the variable MaxSize at any program point after the fixed point computation is at most $N \times k$. Hence, choosing $s' = s + N \times k$ enables the algorithm to discover equivalences among expressions of size s. The proof of Theorem 3 now follows easily from Lemma 4.

3.6 Complexity Analysis

Let j be the number of join points in the program. Let I be the maximum number of iterations of any loop performed by the algorithm. (It follows from Theorem 1 that I is upper bounded by k; however, in practice, this may be a small constant). One join operation $\texttt{Join}(G_1, G_2, s')$ takes time $O(k \times s') = O(k \times (s + N \times k))$. Hence, the total cost of all join operations is $O(k \times (s + N \times k) \times j \times I)$. The cost of all assignment operations is $O(N \times I)$. Hence, the total complexity of the algorithm is dominated by the cost of the join operations (assuming $j \geq 1$). For global value numbering, the choice of $s = N$ suffices, yielding a total complexity of $O(k^2 \times I \times N \times j) = O(k^3 \times N \times j)$ for the algorithm.

4 Programs with Exponential Sized Value Graph Representation for Sets of Herbrand Equivalences

Let m be any positive integer. In this section, we show that there is an acyclic program P_m of size $O(m^2)$ such that any value graph representation of the set of Herbrand equivalences that are true at the end of the program requires $\Theta(2^m)$ size. The program P_m is described in Section 4.2, and is shown in Figure 6. The program P_m involves some non-trivial expressions. To describe these expressions, and to prove that the set of Herbrand equivalences that are true at the end of program P_m requires $\Theta(2^m)$ size, we introduce some notation in Section 4.1.

Let n be the largest integer such that $n \leq m$ and n is a power of 2. Note that $n \geq \frac{m}{2}$.

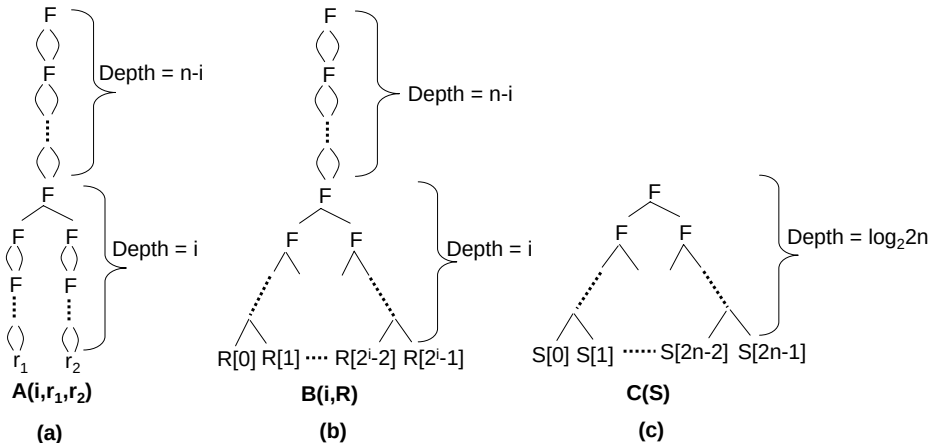

Fig. 4. Value graph representation of expressions $A(i, r_1, r_2)$, $B(i, R)$ and $C(S)$.

4.1 Notation

In this section, we describe some special expressions, sets of expressions, and their properties. For any integer $i \in \{1, \ldots, n\}$ and expressions r_1 and r_2, let $A(i, r_1, r_2)$ denote the expression as shown in Figure 4(a). For any integer $i \in \{1, \ldots, n\}$ and sets of expressions $\tilde{r}_1$ and $\tilde{r}_2$, let $\tilde{A}(i, \tilde{r}_1, \tilde{r}_2)$ denote the following set of expressions:

$$\tilde{A}(i, \tilde{r}_1, \tilde{r}_2) = \{A(i, r_1, r_2) \mid r_1 \in \tilde{r}_1, r_2 \in \tilde{r}_2\}$$

For any integer $i \in \{1, \ldots, n\}$ and an array $R[0 \ldots 2^i - 1]$ of expressions, let $B(i, R)$ denote the expression as shown in Figure 4(b). For any integer $i \in \{1, \ldots, n\}$ and an array $\tilde{R}[0 \ldots 2^i - 1]$ of sets of expressions, let $\tilde{B}(i, \tilde{R})$ denote the following set of expressions:

$$\tilde{B}(i, \tilde{R}) = \{B(i, R) \mid \forall j \in \{0, \ldots, 2^i - 1\}, R[j] \in \tilde{R}[j]\}$$

Using the definitions of $\tilde{A}(i, \tilde{r}_1, \tilde{r}_2)$ and $\tilde{B}(i, \tilde{R})$, we can show that

$$\begin{aligned} \tilde{A}(i+1, \tilde{r}_1, \tilde{r}_2) \cap \tilde{B}(i, \tilde{R}) &= \tilde{B}(i+1, \tilde{R}') \\ \tilde{R}'[j] &= \tilde{R}[j] \cap \tilde{r}_1, \quad 0 \le j < 2^i \\ \tilde{R}'[j] &= \tilde{R}[j - 2^i] \cap \tilde{r}_2, \quad 2^i \le j < 2^{i+1} \end{aligned} \tag{1}$$

Equation 1 is also illustrated diagrammatically in Figure 5. The point to note is that if $\tilde{R}[0], \ldots, \tilde{R}[2^i - 1]$ are all distinct sets of expressions, then the most succinct value graph representation of $\tilde{B}(i, \tilde{R})$ is as shown in Figure 5(b). If $\tilde{r}_1$ and $\tilde{r}_2$ are such that for all $0 \le j_1, j_2 < 2^i$, the sets $\tilde{r}_1 \cap \tilde{R}[j_1]$, $\tilde{r}_2 \cap \tilde{R}[j_2]$ both non-empty and distinct, then the most succinct value graph representation of $\tilde{B}(i, \tilde{R}) \cap \tilde{A}(i+1, \tilde{r}_1, \tilde{r}_2)$ is as shown in Figure 5(c), whose representation is almost double the size of $\tilde{B}(i, \tilde{R})$ (even though it has fewer elements!).

Note that $\tilde{A}(1, \tilde{r}_1, \tilde{r}_2) = \tilde{B}(1, \tilde{R})$ where $\tilde{R}[1] = \tilde{r}_1$ and $\tilde{R}[2] = \tilde{r}_2$. Hence, using Equation 1, we can prove by induction on i that:

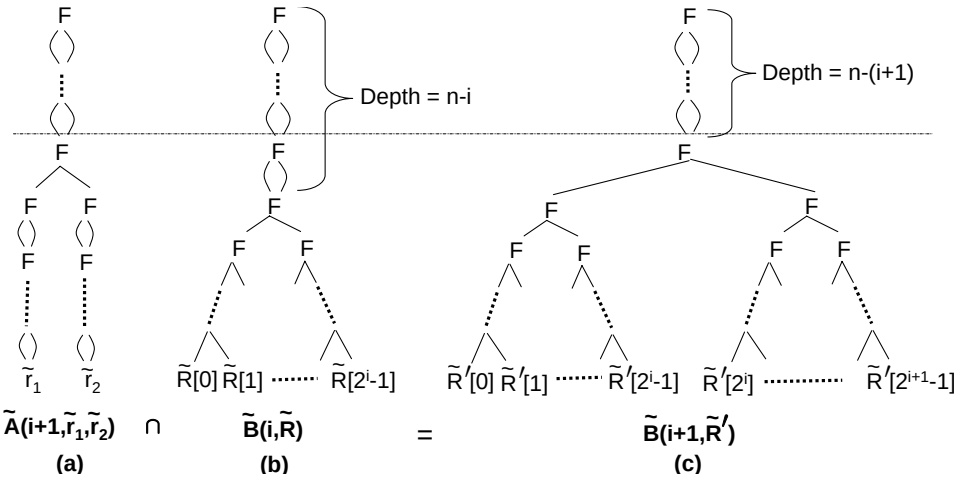

Fig. 5. Relationship between sets $\tilde{A}(i+1, \tilde{r}_1, \tilde{r}_2)$ and $\tilde{B}(i, \tilde{R})$. Nodes immediately below the horizontal dotted line are at the same depth $n - (i + 1)$ from the corresponding root nodes.

Proposition 2. *For any $i \in \{1, .., n\}$, let $r_{i,1}$ and $r_{i,2}$ be some sets of expressions. For any integer j, let $j_n \ldots j_1$ be the binary representation of j. Then,*

$$\bigcap_{i=1}^{n} \tilde{A}(i, \tilde{r}_{i,1}, \tilde{r}_{i,2}) = \tilde{B}(n, \tilde{R})$$

$$\tilde{R}[j] = \bigcap_{i=1}^{n} \tilde{r}_{i,j_i+1}, \quad 0 \leq j < 2^n$$

Our goal is to construct a program P_m such that it satisfies the equivalences $E_i = \{z = e \mid e \in \tilde{A}(i, \tilde{r}_{i,1}, \tilde{r}_{i,2})$ at the i^{th} predecessor of some join point. Note that after the join point it will satisfy the equivalences $E = \{z = e \mid e \in \tilde{B}(n, \tilde{R})\}$, where $\tilde{B}(n, \tilde{R})$ is as defined in Proposition 2. The representation of E would require size exponential in n if the sets $\tilde{r}(i, 1)$ and $\tilde{r}(i, 2)$ are such that for each distinct choice of bits $j_1, .., j_n$, the set $\bigcap_{i=1}^{n} \tilde{r}_{i,j_i+1}$ is distinct and non-empty. This can be easily accomplished if the program has 2^n variables (by choosing sets $\tilde{r}(i, 1)$ and $\tilde{r}(i, 2)$ to contain an appropriate subset of the 2^n program variables such that $\bigcap_{i=1}^{n} \tilde{r}_{i,j_i+1}$ is a singleton set containing a distinct program variable). In the rest of this section, we show how to accomplish this using just n program variables (by choosing sets $\tilde{r}(i, 1)$ and $\tilde{r}(i, 2)$ to contain small terms constructed from just n program variables).

For any array $S[0\,..\,2n-1]$ of expressions, let $C(S)$ denote the expression as shown in Figure 4(c). For any array $\tilde{S}[0\,..\,2n-1]$ of sets of expressions, let $\tilde{C}(\tilde{S})$ denote the following set of expressions:

$$\tilde{C}(\tilde{S}) = \{C(S) \mid \forall i \in \{0, .., 2n-1\}, S[i] \in \tilde{S}[i]\}$$

For any integer $i \in \{1,..,n\}$, $b \in \{1,2\}$, let $S_{i,b}[0..2n-1]$ be the following array of expressions,

$$\begin{aligned} S_{i,b}[j] &= 1, \text{ if } j = 2(i-1)+b-1 \\ &= 0, \text{ otherwise} \end{aligned}$$

For any integer $i \in \{1,..,n\}$, $b \in \{1,2\}$, let $\tilde{S}_{i,b}[0..2n-1]$ be the following array of sets of expressions,

$$\begin{aligned} \tilde{S}_{i,b}[j] &= \{x_i, 1\}, \text{ if } j = 2(i-1)+b-1 \\ &= \{x_1,..,x_{i-1},x_{i+1},..,x_n,0\}, \text{ otherwise} \end{aligned}$$

For any integer $j \in \{0,..,2^n-1\}$, let $j_n..j_1$ be the binary representation of j. Let $T_j[0..2n-1]$ be the following array of expressions:

$$\begin{aligned} T_j[2(\ell-1)+j_\ell] &= x_\ell, \quad 0 \le \ell < n \\ T_j[2(\ell-1)+1-j_\ell] &= 0, \quad 0 \le \ell < n \end{aligned}$$

Using the definitions of $\tilde{C}(\tilde{S})$, $\tilde{S}_{i,b}$ and $\tilde{T}_j$), we can prove the following proposition.

Proposition 3. *Let $j \in \{0,..,2^n-1\}$. Let $j_n..j_1$ be the binary representation of j. Then,*

$$\bigcap_{i=1}^{n} \tilde{C}(\tilde{S}_{i,j_i+1}) = \{C(T_j)\}$$

Note that $\tilde{C}(\tilde{S}_{i,b})$ are an appropriate choice for sets $\tilde{r}_{i,b}$. The following proposition, which follows from Proposition 2 and Proposition 3, summarizes the interesting property of these sets.

Proposition 4. *For any $i \in \{1,..,n\}$, Then,*

$$\begin{aligned} \bigcap_{i=1}^{n} \tilde{A}(i, \tilde{C}(\tilde{S}_{i,1}), \tilde{C}(\tilde{S}_{i,2})) &= \{B(n,R)\} \\ R[j] &= C(T_j), \quad 0 \le j < 2^n \end{aligned}$$

4.2 The Program P_m

The program P_m, which contains an n-branch switch statement, is shown in Figure 6. It consists of $n+1$ local variables, $z, x_1, x_2, .., x_n$. The expressions a_i and b are defined below.

$$\begin{aligned} a_i &= A(i, C(S_{i,1}), C(S_{i,2})) \\ b &= B(n,R) \\ R[j] &= C(T_j), \quad 0 \le j < 2^n \end{aligned}$$

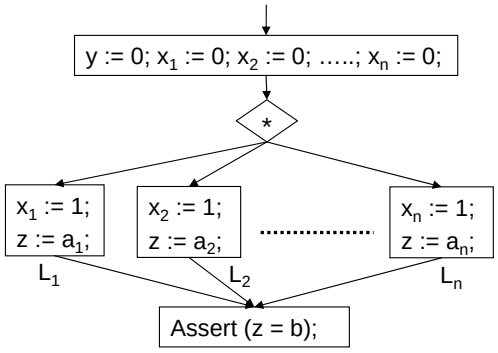

Fig. 6. The program P_m. The expressions a_i and b are defined in Section 4.2.

Note that for all $i \in \{1, .., n\}$, $size(a_i) \leq 6n$. Thus, the size of program P_m is $O(n^2) = O(m^2)$. We now show that any value graph representation of the set of equivalences that hold at the end of the program P_m requires $\Theta(2^m)$ nodes. First note that it is important to maintain only equivalences of the form $x = e$ where x is a variable and e an expression. (This also follows from the fact that the SED data structure that we introduce in Section 3.1 can represent the set of equivalences at any program point). The following theorem implies that there is only one such equivalence, namely $z = b$, that holds at the end of program P_m.

Theorem 4. *Let E denote the set of all Herbrand equivalences of the form $x = e$ that are true at the end of the program P_m. Then,*

$$E = \{z = b\}$$

Proof. Let E_i denote the set of all Herbrand equivalences of the form $x = e$ that are true at point L_i in the program P_m. Then it is not difficult to see that:

$$E_i = \{z = e \mid e \in \tilde{A}(i, \tilde{C}(\tilde{S}_{i,1}), \tilde{C}(\tilde{S}_{i,2}))\} \cup \{x_i = 1\} \cup \{x_j = 0 \mid 1 \leq j \leq n, j \neq i\}$$

Using Proposition 4 we get:

$$\begin{aligned} E = \bigcap_{i=1}^{n} E_i &= \{z = e \mid e \in \bigcap_{i=1}^{n} \tilde{A}(i, \tilde{C}(\tilde{S}_{i,1}), \tilde{C}(\tilde{S}_{i,2}))\} \\ &= \{z = e \mid e \in \{b\}\} = \{z = b\} \end{aligned}$$

Note that any value graph representation of expression b must have size $\Theta(2^n)$ since $R[j_1] \neq R[j_2]$ for $j_1 \neq j_2$. Hence, any value graph representation of the equivalence $z = b$ requires $\Theta(2^n) = \Theta(2^m)$ nodes.

5 Related Work

Kildall's Algorithm: Kildall's algorithm [7] performs an abstract interpretation over the lattice of sets of Herbrand equivalences. It represents the set of Herbrand equivalences at each program point by means of a structured partition.

The join operation for two structured partitions π_1 and π_2 is defined to be their intersection. Kildall's algorithm is complete in the sense that if it terminates, then the structured partition at any program point reflects all Herbrand equivalences that are true at that point. However, the complexity of Kildall's algorithm is exponential. The number of elements in a partition, and the size of each element in a partition can all be exponential in the number of join operations performed.

Alpern, Wegman and Zadeck's (AWZ) Algorithm: The AWZ algorithm [1] works on the value graph representation [8] of a program that has been converted to SSA form. A value graph can be represented by a collection of nodes of the form $\langle V, t\rangle$ where V is a set of variables, and the type t is either $\perp$, a constant c (indicating that the node has no successors), $F(n_1, n_2)$ or $\phi_m(n_1, n_2)$ (indicating that the node has two ordered successors n_1 and n_2). ϕ_m denotes the ϕ function associated with the m^{th} join point in the program. Our data structure SED can be regarded as a special form of a value graph which is acyclic and has no ϕ-type nodes. The main step in the AWZ algorithm is to use congruence partitioning to merge some nodes of the value graph.

The AWZ algorithm cannot discover all equivalences among program terms. This is because it treats ϕ functions as uninterpreted. The ϕ functions are an abstraction of the if-then-else operator wherein the conditional in the if-then-else expression is abstracted away, but the two possible values of the if-then-else expression are retained. Hence, the ϕ functions satisfy the following two equations.

$$\forall e: \quad \phi_m(e, e) = e \tag{2}$$

$$\forall e_1, e_2, e_3, e_4: \quad \phi_m(F(e_1, e_2), F(e_3, e_4)) = F(\phi_m(e_1, e_3), \phi_m(e_2, e_4)) \tag{3}$$

Rüthing, Knoop and Steffen's (RKS) Algorithm: Like the AWZ algorithm, the RKS algorithm [12] also works on the value graph representation of a program that has been converted to SSA form. It tries to capture the semantics of ϕ functions by applying the following rewrite rules, which are based on equations 2 and 3, to convert program expressions into some normal form.

$$\langle V, \phi_m(n, n)\rangle \text{ and } n \;\rightarrow\; \langle V \cup \mathit{Vars}(n), \mathit{Type}(n)\rangle \tag{4}$$

$$\langle V, \phi_m(\langle V_1, F(n_1, n_2)\rangle, \langle V_2, F(n_3, n_4)\rangle)\rangle \;\rightarrow\; \langle V, F(\langle \emptyset, \phi_m(n_1, n_3)\rangle, \langle \emptyset, \phi_m(n_2, n_4)\rangle)\rangle \tag{5}$$

Nodes on the left of the rewrite rules are replaced by the (new) node on the right, and incoming edges to nodes on the left are made to point to the new node. However, there is a precondition to applying the second rewriting rule.

$$P: \forall \text{ nodes } n \in \mathit{succ}^*(\{\langle V_1, F(n_1, n_2)\rangle, \langle V_2, F(n_3, n_4)\rangle\}),\ \mathit{Vars}(n) \neq \emptyset$$

The RKS algorithm assumes that all assignments are of the form $x := F(y, z)$ to make sure that for all original nodes n in the value graph, $\mathit{Vars}(n) \neq \emptyset$. This precondition is necessary in arguing termination for this system of rewrite rules,

and proving the polynomial complexity bound. The RKS algorithm alternately applies the AWZ algorithm and the two rewrite rules until the value graph reaches a fixed point. Thus, the RKS algorithm discovers more equivalences than the AWZ algorithm.

The RKS algorithm cannot discover all equivalences even in acyclic programs. This is because the precondition P can prevent two equal expressions from reaching the same normal form. On the other hand lifting precondition P may result in the creation of an exponential number of new nodes, and an exponential number of applications of the rewrite rules. Such would be the case when, for example, the RKS algorithm is applied to the program P_m described in Section 4.

The RKS algorithm has another problem, which the authors have identified. It fails to discover all equivalences in cyclic programs, even if the precondition P is lifted. This is because the graph rewrite rules add a degree of pessimism to the iteration process. While congruence partitioning is optimistic, it relies on the result of the graph transformations which are pessimistic, as they are applied outside of the fixed point iteration process.

Gulwani and Necula's Randomized Algorithm: Recently, we gave a *randomized* polynomial-time algorithm that discovers all Herbrand equivalences among program terms [6]. This algorithm can also verify all Herbrand equivalences that are true at any point in a program. However, there is a small probability (over the choice of the random numbers chosen by the algorithm) that this algorithm deduces false equivalences. This algorithm is based on the idea of random interpretation, which involves performing abstract interpretation using randomized data structures and algorithms.

6 Conclusion and Future Work

We have given a polynomial-time algorithm for global value numbering. We have shown that there are programs for which the set of all equivalences contains terms whose value graph representation requires exponential size. This justifies the design of our algorithm, which discovers all equivalences among terms of size at most s in time that grows linearly with s.

An interesting theoretical question is to figure if there exist representations that may avoid the exponential lower bound for representing the set of all Herbrand equivalences.

The next step is to perform experiments to compare the different algorithms with regard to running time and number of equivalences discovered. Results of our algorithm can also be used as a benchmark to estimate the incompleteness of the existing algorithms.

An interesting direction of future work is to extend this algorithm to perform precise inter-procedural value numbering. It would also be useful to extend the algorithm to reason about some properties of program operators like commutativity, associativity or both.

Acknowledgments

This research was supported in part by the National Science Foundation Grants CCR-9875171, CCR-0085949, CCR-0081588, CCR-0234689, CCR-0326577, CCR-00225610, and gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11. ACM, 1988.
2. C. Click. Global code motion/global value numbering. In *Proccedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 246–257, June 1995.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.
4. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1990.
5. K. Gargi. A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, volume 37, 5, pages 45–56. ACM Press, June 17–19 2002.
6. S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *31st Annual ACM Symposium on POPL*. ACM, Jan. 2004.
7. G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on Principles of Programming Language*, pages 194–206, Oct. 1973.
8. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.
9. G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 83–94. ACM SIGPLAN, June 2000.
10. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference*, volume LNCS 1384, pages 151–166. Springer, 1998.
11. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM, 1988.
12. O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 1999.
13. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.