Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 64 (2007) 97–114
Science of Computer Programming
www.elsevier.com/locate/scico

A polynomial-time algorithm for global value numbering

Sumit Gulwani [a,*], George C. Necula [b]

[a] *Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA*
[b] *Department of Computer Science, UC-Berkeley, Berkeley, CA 94720, USA*

Received 31 January 2005; received in revised form 15 October 2005; accepted 15 March 2006
Available online 9 October 2006

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. We provide a detailed analytical comparison of some of these algorithms.

Keywords: Global value numbering; Uninterpreted functions; Abstract interpretation; Herbrand equivalences

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 [17], common sub-expression elimination, invariant code motion [2,14], induction variable elimination, branch elimination, branch fusion, and loop jamming [9]. 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 [13,12], 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* [16]. The process of discovering such a 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.

* Corresponding author.
E-mail addresses: sumitg@microsoft.com (S. Gulwani), necula@cs.berkeley.edu (G.C. Necula).
URLs: http://research.microsoft.com/~sumitg (S. Gulwani), http://www.cs.berkeley.edu/~necula (G.C. Necula).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2006.03.005

Existing deterministic algorithms for global value numbering are either too expensive or imprecise. The precise algorithms are based on an early algorithm by Kildall [8], 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 [16]. 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, but at the price of being significantly less precise than Kildall's algorithm. The novel idea in the 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) [16] 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 algorithms both use a data structure called the value graph [9], 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 also proposed a set of balanced algorithms that are efficient, but they too are 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 for answering questions about non-program terms, but it also allows forward 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, and 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 that can arise at any program point in our abstracted program model (which only allows non-deterministic conditionals)has finite height k, where k is the number of program variables. We prove this result in Section 3.6. 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 the presence of loops by adding a degree of pessimism. This leads to incompleteness in the presence of loops, as is the case with the RKS algorithm [16]. Instead, our algorithm is based on an abstract interpretation, similar to Kildall's algorithm, while using a more sophisticated join operation. Note that even though the lattice of Herbrand equivalences has small height, representing the lattice elements and performing lattice operations on them can take exponential time and space, as pointed out in the first observation above. We avoid this problem by maintaining a bounded size approximation of lattice elementsthat is sufficient to discover all Herbrand equivalences of bounded size. 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.

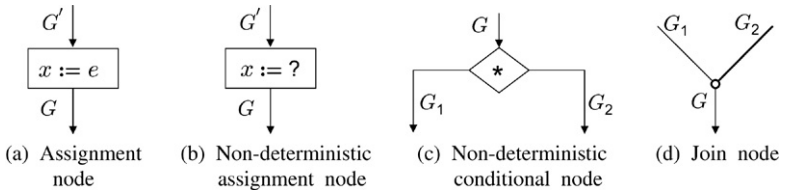


Fig. 1. Flowchart nodes.

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, we can model any uninterpreted function F^a of any constant arity a using the given binary uninterpreted function F by employing the following closure trick:

$$F^a(e_1, \ldots, e_a) = F(e_1, e'_2), \text{ where } e'_i = \begin{cases} F(e_i, e'_{i+1}) & \text{for } 2 \leq i \leq a-1 \\ F(e_a, x_{F^a}) & \text{for } i = a. \end{cases}$$

Here x_{F^a} is a fresh variable (which can be regarded as a new input variable) associated with the uninterpreted function F^a. If we regard a to be a constant, then this modeling does not alter the quantities on which the computational complexity of the algorithm depends (except by a constant factor).

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 a 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 $e_1 = e_2$, where $\mathit{size}(e_1) \leq s$ and $\mathit{size}(e_2) \leq s$. 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 the flowchart nodes shown in Fig. 1 using the transfer functions described in Section 3.2 through Section 3.5. In the presence of loops, it goes around loops until a fixed point is reached, as described in Section 3.6.

3.1. Notation and data structure (SED)

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 η 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 $F(\eta_1, \eta_2)$, indicating that the node has two ordered successors η_1 and η_2.

In any SED G, for every variable x, there is exactly one node $\langle V, t\rangle$, denoted by $\mathit{Node}_G(x)$, such that $x \in V$. For every type t that is not $\perp$, there is at most one node of that type. We use the notation $\mathit{Node}_G(c)$ to refer to the node with type c. For any SED node η, we use the notation $\mathit{Vars}(\eta)$ to denote the set of variables labeling node η, and

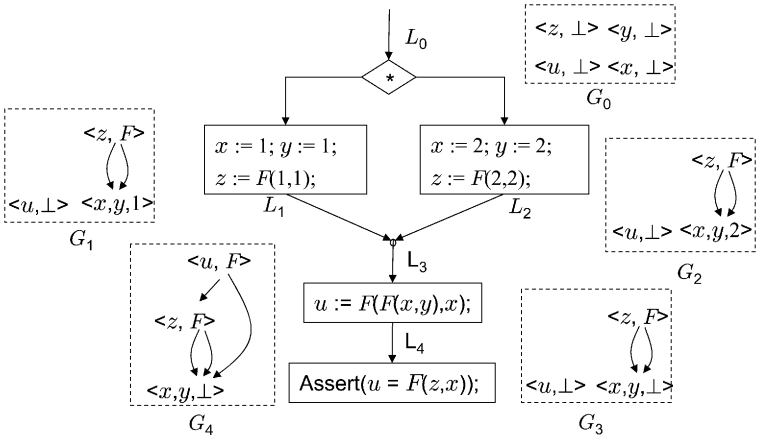


Fig. 2. This figure shows a program and the SEDs computed by our algorithm at various program points. G_i, shown in dotted box, represents the SED at program point L_i.

$Type(\eta)$ to denote the type of node η. Every node η in an SED represents the following set of terms $Terms(\eta)$, which are all known to be equivalent:

$$\begin{aligned} Terms(V, \bot) &= V \\ Terms(V, c) &= V \cup \{c\} \\ Terms(V, F(\eta_1, \eta_2)) &= V \cup \{F(e_1, e_2) \mid e_1 \in Terms(\eta_1), e_2 \in Terms(\eta_2)\}. \end{aligned}$$

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.

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

In figures showing SEDs, we omit the set delimiters "{" and "}", and represent a node $\langle\{x_1, \ldots, x_m\}, t\rangle$ as $\langle x_1, \ldots, x_m, t\rangle$. Fig. 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.

We now describe an alternative representation for SED that is useful in understanding the join algorithm, and for proving the fixed point result. An SED can be represented by a partition of all program variables into equivalence classes (where all variables in an equivalence class are known to be equal). Furthermore, some of these equivalences classes are constrained to be equal to some F-term over the values of other equivalence classes. The sets of variables V in nodes of an SED represent these equivalence classes, and the type of those nodes represent the F-term that the corresponding equivalence class is constrained to be equal to. For example, the SED G_4 shown in Fig. 2 can be represented by the following partition of variables: $\{u\}$, $\{z\}$, and $\{x, y\}$. The equivalence class $\{u\}$ is constrained to be equal to $F(z, x)$, and the equivalence class $\{z\}$ is constrained to be equal to $F(x, x)$. The equivalence class $\{x, y\}$ is unconstrained.

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

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

The SED at other program points are computed from the SEDs at previous program points by using the transfer functions described in the following subsections. These transfer functions may yield SEDs with unnecessary nodes, which may be removed. A node is unnecessary when all its ancestor nodes or all its descendant nodes have an empty set of variables. The removal of unnecessary nodes can result in dangling pointers for the types of some (necessary) nodes. The types of such nodes should be set to $\bot$.

3.2. Assignment node

See Fig. 1(a). 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 `Assignment` function. SED G_4 in Fig. 2 shows an example of the `Assignment` function.

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

  GetNode(G, e) =
1   match e with
2     y: return Node_G(y);
3     F(e1, e2): let η1 = GetNode(G, e1) and η2 = GetNode(G, e2) in
4               if ⟨V, F(η1, η2)⟩ ∈ G for some V, return ⟨V, F(η1, η2)⟩;
5               else G := G ∪ ⟨∅, F(η1, η2)⟩; return ⟨∅, F(η1, η2)⟩;
```

`GetNode`(G, e) returns a node η such that $e \in \mathit{Terms}(\eta)$ (and in the process possibly extends G) in $O(\mathit{size}(e))$ time. `ReplaceVars`(G, η, V) replaces the set of variables in node η by V (in place) in SED G. Lines 4 and 5 in `Assignment` function move variable x to the node `GetNode`(G, e) to reflect the equivalence $x = e$. Hence, the following lemma holds.

Lemma 1 (*Soundness and Completeness of* `Assignment`). *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. Non-deterministic assignment node

See Fig. 1(b). If the SED G' before a non-deterministic assignment node is $\bot$, then the SED G after the non-deterministic assignment node is also $\bot$. Otherwise, the SED G after a non-deterministic assignment node $x :=?$ is obtained from SED G' using the following function, which removes variable x from $\mathit{Node}_{G'}(x)$, and creates a new node $\langle\{x\}, \top\rangle$.

```
  Non-det-Assignment(G', x :=?) =
1   G := G';
2   let ⟨V, t⟩ = Node_G(x) in
3   ReplaceVars(G, ⟨V, t⟩, V − {x});
4   G := G ∪ {⟨{x}, ⊤⟩};
5   return G;
```

The following lemma holds.

Lemma 2 (*Soundness and Completeness of* `Non-det-Assignment`). *Let* $G =$ `Non − det − Assignment`$(G', x :=?)$. *Let* e_1 *and* e_2 *be two expressions. Let* $e'_1 = e_1[^{x'}/_{x}]$ *and* $e'_2 = e_2[^{x'}/_{x}]$ *for some fresh variable* x' *that does not occur in* e_1 *and* e_2. *Then,* $G \models e_1 = e_2$ *iff* $G' \models e'_1 = e'_2$.

3.4. Non-deterministic conditional node

See Fig. 1(c). The SEDs G^1 and G^2 on the two branches of a non-deterministic conditional node are simply a copy of the SED G before the non-deterministic conditional node.

3.5. Join node

See Fig. 1(d). 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 the 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 3.7). Fig. 2 shows an example of the `Join` function.

For any SED G, let $\prec_G$ denote a partial ordering on the 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 $x \in Variables(F(e_1, e_2))$.

```
Join(G1,G2,s') =
1   for all nodes η1 ∈ G1 and η2 ∈ G2, memoize[η1, η2] := undefined;
2   G := ∅;
3   for each variable x ∈ T in the order ≺G1 do
4       counter := s';
5       Intersect(Node_G1(x), Node_G2(x));
6   return G;

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

The `Join` function is similar to the finite automata intersection algorithm. It is easier to understand the `Join` function by ignoring the use of the *counter* variable, which is introduced for efficiency reasons rather than correctness. If we ignore the use of *counter* variable, then $Join(G_1, G_2, s')$ returns an SED G such that G implies all equivalences that are implied by both G_1 and G_2. In that case however, the size of G as well as the computational complexity of the `Join` function will be quadratic in the size of G_1 and G_2. Hence a join of n SEDs may result in an SED whose size is exponential in the size of the input SEDs. (This would be the case, for example, for the program shown in Fig. 5.)

The use of a *counter* variable produces a *pruned* version of G that maintains all equivalences of size at most s' (as stated formally in Lemma 4). The *pruned* version of G represents the SED that can be obtained from G by removing constraints of those equivalence classes represented by G (recall the alternative representation of SEDs as discussed in Section 3.1) that are of size greater than s'. Computing a pruned version of G as opposed to G itself is sufficient, since we are interested in computing equivalences of bounded size rather than all equivalences. The use of a *counter* variable thus ensures that the call to `Intersect` function in `Join` terminates in $O(s')$ time. Hence, the complexity of the `Join` function with use of a *counter* variable is $O(s' \times k)$. An alternative would have been to compute G by running the `Join` function without the use of a *counter* variable, and then pruning G. This would, however have an increased computational complexity of $O(s'^2)$.

The following proposition describes the property of `Intersect` function that is required to prove the correctness of the `Join` function (Lemma 4).

Proposition 3. *Let $\eta_1 = \langle V_1, t_1\rangle$ and $\eta_2 = \langle V_2, t_2\rangle$ be any nodes in SEDs G_1 and G_2 respectively. Let $n = \langle V, t\rangle = \texttt{Intersect}(\eta_1, \eta_2)$. Suppose that $n \neq \langle \emptyset, \bot\rangle$; hence the function $\texttt{Intersect}(\eta_1, \eta_2)$ adds the node n to G. Let α be the value of the counter variable when $\texttt{Intersect}(\eta_1, \eta_2)$ is first called. Then,*

(P1) $Terms(\eta) \subseteq Terms(\eta_1) \cap Terms(\eta_2)$.
(P2) $Terms(\eta) \supseteq \{e \mid e \in Terms(\eta_1), e \in Terms(\eta_2), size(e) \leq \alpha\}$.

The proof of Proposition 3 is by induction on the sum of height of nodes η_1 and η_2 in G_1 and G_2 respectively. We sketch a brief outline of the proof here; the detailed proof is given in Appendix A.1. Claim P1 follows from the observation that $t = F(...)$ or c only if both t_1 and t_2 are $F(...)$ or c respectively (lines 7 and 8), and $V = V_1 \cap V_2$ (line 10). Claim P2 relies on bottom-up processing of one of the SEDs (line 3 in `Join` function), and memoization of calls to the `Intersect` function (line 12). Let e' be one of the *smallest* expressions (in terms of $size$) such that $e' \in Terms(\eta_1) \cap Terms(\eta_2)$. 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 $e' \in Terms(\eta)$.

Lemma 4 (*Soundness and Completeness of* `Join`). *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 4 follows from Proposition 3 and the definition of $\models$.

3.6. 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 [9]).

Theorem 1 (*Fixed Point Theorem*). *Let $G_1, \ldots, G_\ell$ be the SEDs computed by the algorithm at some program point inside a loop in successive iterations of that loop such that G_{i+1} implies a strictly smaller subset of equivalences than those implied by G_i. Then, $\ell \leq k + 1$, where k is the number of program variables.*

Proof. Consider the alternative representation of SEDs in terms of partitions of constrained or unconstrained equivalence classes of the program variables (as discussed in Section 3.1). Now observe that G_i can be obtained from G_{i+1} only by constraining an unconstrained equivalence class or by merging an unconstrained equivalence class with another (constrained or unconstrained) equivalence class. Hence, the number of unconstrained equivalence classes in G_i is strictly smaller than in G_{i+1}. Since the number of unconstrained equivalence classes in G_ℓ can be at most k, the result follows.

3.7. Correctness of the algorithm

The correctness of the algorithm follows from Theorems 2 and 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), non-det-assignment operation (Lemma 2 in Section 3.3) and the join operation (Lemma 4 in Section 3.5).

Theorem 3 (*Completeness Theorem*). *Let $e_1 = e_2$ be an equivalence that holds at a program point P such that $size(e_1) \leq s$ and $size(e_2) \leq s$. 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 the 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 Fig. 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.

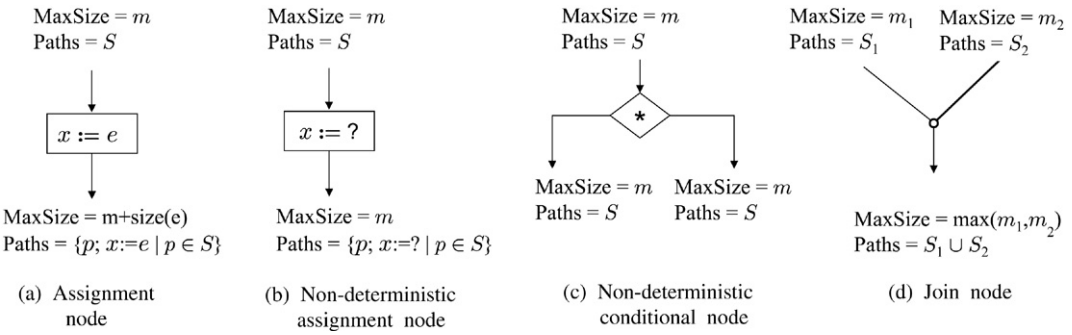


Fig. 3. Flowchart nodes.

Lemma 5. *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, $G \models e_1 = e_2$.*

The proof of Lemma 5 is by induction on the number of operations performed by the algorithm, and is given in Appendix A.2.

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 5.

3.8. 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. We first describe the program P_2 and then show how to generalize it to obtain the program P_m.

The program P_2 is shown in Fig. 4. First note that the assertion $z = b$ at the end of the program is true. Also, note that $size(b) \approx size(a_1) \times size(a_2)$. It is not difficult to see that $z = b$ is the only non-trivial equivalence that holds at the end of the program. Hence, the size of the value flow graph representation of the set of equivalences that hold at the end of the program is $\Theta(size(b)) = \Theta(size(a_1) \times size(a_2))$, while the program size is $O(size(a_1) + size(a_2))$.

We now describe the program P_m. Let n be the largest integer such that $n \leq m$ and n is a power of 2. (Note that $n \geq \frac{m}{2}$.) The program P_m, which contains an n-branch switch statement, is shown in Fig. 5. It consists of $n + 1$ local variables: $z, x_1, x_2, \ldots, x_n$, and uses expressions a_i and b, which are defined below.

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

For any integer $i \in \{1, \ldots, n\}$ and expressions r_1 and r_2, $A(i, r_1, r_2)$ denotes the expression as shown in Fig. 6(a). For any integer $i \in \{1, \ldots, n\}$ and an array $R[0 \ldots 2^i - 1]$ of expressions, $B(i, R)$ denotes the expression as shown

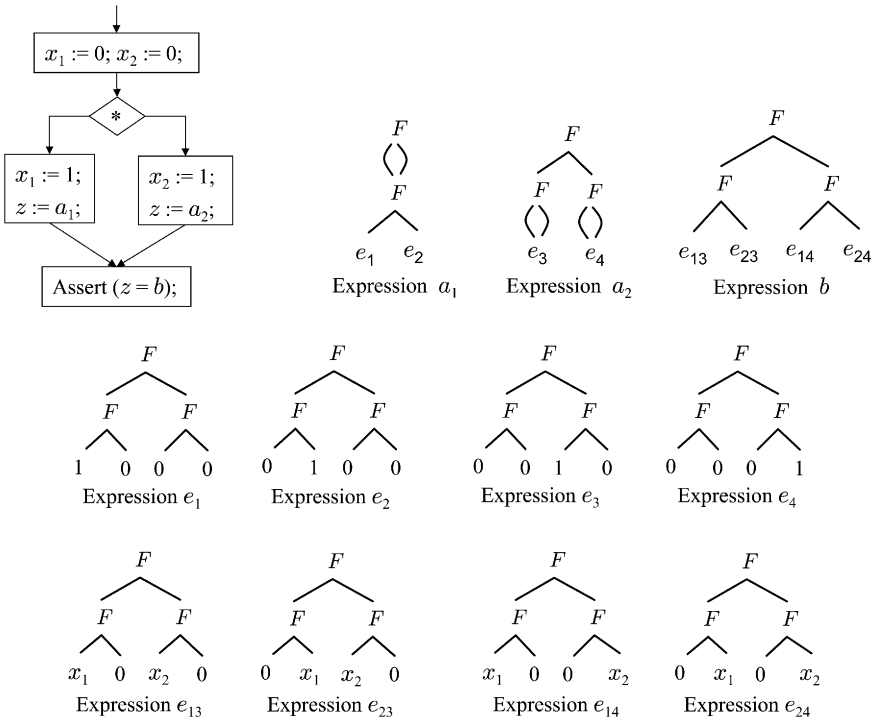


Fig. 4. The program P_2.

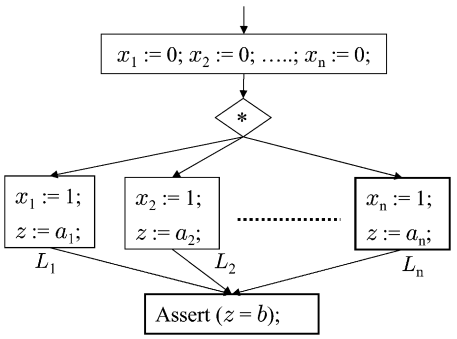


Fig. 5. The program P_m. Expressions a_i and b are as defined in Section 4.

in Fig. 6(b). For any array $S[0\,..\,2n-1]$ of expressions, $C(S)$ denotes the expression as shown in Fig. 6(c). For any integer $i \in \{1, \ldots, n\}$, $b \in \{1, 2\}$, $S_{i,b}[0\,..\,2n-1]$ denotes 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 $j \in \{0, \ldots, 2^n-1\}$, let $j_n \ldots j_1$ be the binary representation of j. Then, $T_j[0\,..\,2n-1]$ denotes the following array of expressions:

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

Note that for all $i \in \{1, \ldots, n\}$, $size(a_i) \le 6n$. 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 holds at the end of the program P_m requires $\Theta(2^m)$ nodes. First note that it is sufficient 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

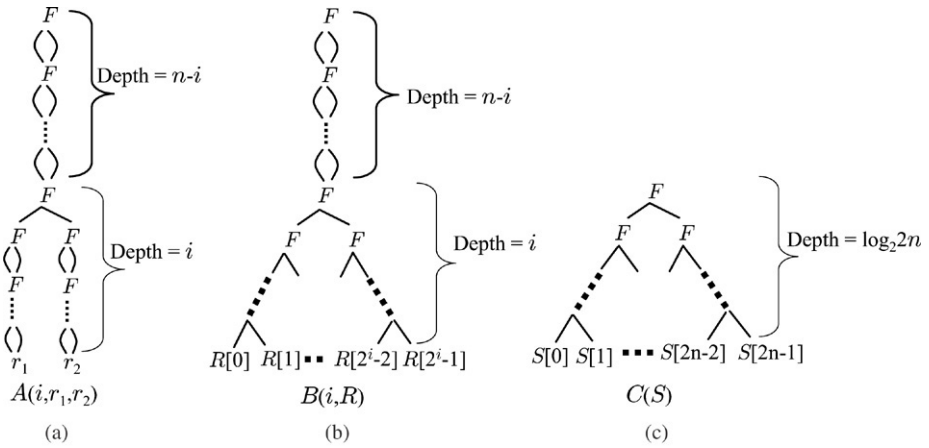


Fig. 6. Value graph representation of expressions $A(i, r_1, r_2)$, $B(i, R)$ and $C(S)$.

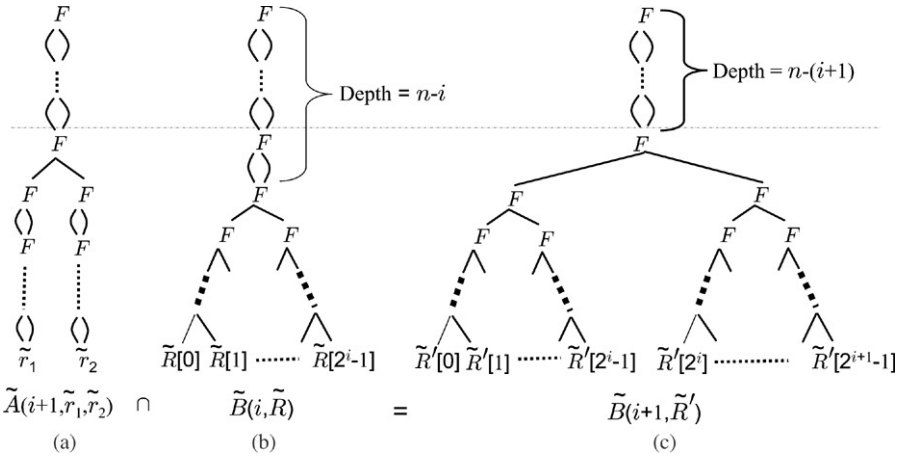


Fig. 7. Relationship between sets $\tilde{A}(i+1, \tilde{r}_1, \tilde{r}_2)$ and $\tilde{B}(i, \tilde{R})$. Nodes immediately below the horizontal dotted line are at the same depth $n-(i+1)$ from the corresponding root nodes.

the set of equivalences at any program point.) Theorem 4 stated below implies that there is only one such equivalence, namely $z = b$, that holds at the end of the program P_m. 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.

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, $E = \{z = b\}$.*

In the remainder of this section, we prove Theorem 4. For this purpose, we first introduce some notation.

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 $\tilde{R}[0 \ldots 2^i - 1]$ 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

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

Eq. (1) is also illustrated diagrammatically in Fig. 7. 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 Fig. 7(b). If $\tilde{r}_1$ and $\tilde{r}_2$ are such that for all $0 \le j_1, j_2 < 2^i$, the sets $\tilde{r}_1 \cap \tilde{R}[j_1]$, $\tilde{r}_2 \cap \tilde{R}[j_2]$ are 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 Fig. 7(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 Eq. (1), we can prove by induction on i that:

Proposition 6. *For any $i \in \{1, \ldots, n\}$, let $\tilde{r}_{i,1}$ and $\tilde{r}_{i,2}$ be some sets of expressions. For any integer j, let $j_n \ldots j_1$ be the binary representation of j. Then,*

$$\bigcap_{i=1}^{n} \tilde{A}(i, \tilde{r}_{i,1}, \tilde{r}_{i,2}) = \tilde{B}(n, \tilde{R}), \text{ where } \tilde{R}[j] = \bigcap_{i=1}^{n} \tilde{r}_{i,j_i+1} \text{ for } 0 \le j < 2^n.$$

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

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

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

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

Using the above definitions, we can prove the following proposition.

Proposition 7. *Let $j \in \{0, \ldots, 2^n - 1\}$. Let $j_n \ldots j_1$ be the binary representation of j. Then,*

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

The following proposition, which follows from Propositions 6 and 7, summarizes an interesting property of these sets.

Proposition 8.

$$\bigcap_{i=1}^{n} \tilde{A}(i, \tilde{C}(\tilde{S}_{i,1}), \tilde{C}(\tilde{S}_{i,2})) = \{B(n, R)\}, \text{ where } R[j] = C(T_j), \text{ for } 0 \le j < 2^n.$$

We now prove Theorem 4 using Proposition 8.

Proof (*Theorem* 4). 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:

$$\begin{aligned} E_i &= \{z = e \mid e \in \tilde{A}(i, \tilde{C}(\tilde{S}_{i,1}), \tilde{C}(\tilde{S}_{i,2}))\} \cup \\ &\quad \{x_i = 1\} \cup \{x_j = 0 \mid 1 \le j \le n, j \ne i\}. \end{aligned}$$

Using Proposition 8 we get:

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

5. Related work

In this section, we describe some other algorithms for global value numbering. We provide a detailed analytical comparison of these algorithms. This explains why these algorithms were not able to solve the problem described in this paper in polynomial time.

5.1. Kildall's algorithm

Kildall's algorithm [8] 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 transfer function `Assignment` for an assignment node $x := e$ is:

$$\texttt{Assignment}(\pi) = \{(e_1, e_2) \mid (e_1[^{e}/_{x}], e_2[^{e}/_{x}] \in \pi\}.$$

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. Also, Kildall did not prove any upper bound on the number of iterations required for achieving a fixed point.

Our algorithm is also based on an abstract interpretation. We have proved that the number of iterations required for reaching a fixed point is bounded above by the number of variables live at any point in the program. We avoid the problem of exponential sized representation for equivalences by using a different data structure SED, and a more sophisticated join algorithm:

- Our data structure represents only those partition classes explicitly that have at least one variable. Furthermore, our data structure represents an exponential number of elements in each partition class succinctly by means of DAGs in which the common substructures are shared. This avoids the problem of explicitly maintaining an exponential number of partition classes, and an exponential number of terms in each partition class. This observation was also made by Rüthing, Knoop and Steffen [15,16].
- Kildall's join algorithm is polynomial in the number of terms in the two partition classes whose join is computed, which can be exponential in the value graph representation of the partition classes. Our join algorithm runs in polynomial time in the value graph representation of the partition classes.
- Some elements in a partition class can have an exponential size even if the elements are represented using a value graph representation. Section 4 describes such an example. We get around this problem by maintaining only those terms in each partition class that have size less than $s + N \times k$, where s is a parameter of the algorithm. We prove that this is sufficient to preserve relationships between program terms of size less than s.

5.2. Alpern, Wegman and Zadeck's (AWZ) algorithm

The AWZ algorithm [1] works on the value graph representation [9] 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), $F(\eta_1, \eta_2)$ or $\phi_j(\eta_1, \eta_2)$ (indicating that the node has two ordered successors η_1 and η_2). ϕ_j denotes the ϕ function associated with the jth 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.

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

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

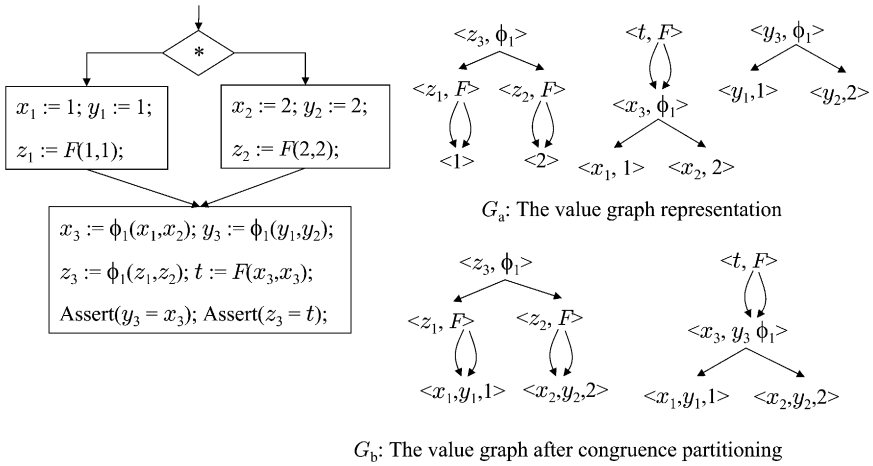


Fig. 8. A program in SSA form, its value graph representation, and the value graph after congruence partitioning. The AWZ algorithm can deduce the first assertion $x_3 = z_3$, but not the second assertion $t = y_3$.

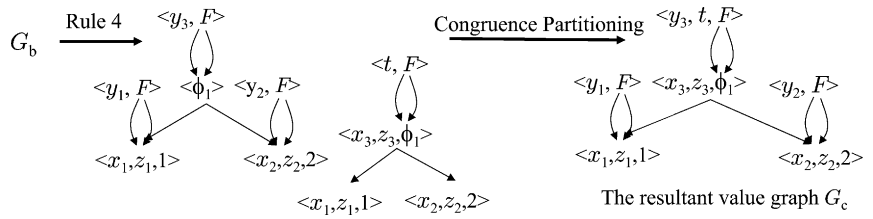


Fig. 9. The value graph for the program in Fig. 8 that results after applying the RKS algorithm. The RKS algorithm can deduce both the assertions $y_3 = x_3$ and $z_3 = t$.

Fig. 8 shows a program for which the AWZ algorithm fails to discover some equivalences. The AWZ algorithm can deduce that $y_3 = x_3$, but it cannot deduce that $z_3 = t$ because it treats ϕ functions as uninterpreted.

5.3. Rüthing, Knoop and Steffen's (RKS) algorithm

Like the AWZ algorithm, the RKS algorithm [16] 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 Eqs. (2) and (3), to convert program expressions into some normal form.

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

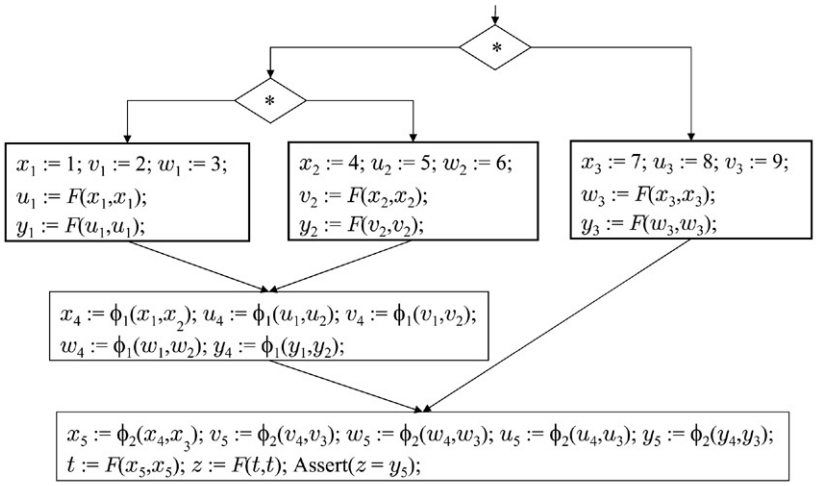
$$\langle V, \phi_j(\langle V_1, F(\eta_1, \eta_2)\rangle, \langle V_2, F(n_3, n_4)\rangle)\rangle \;\rightarrow\; \langle V, F(\langle \emptyset, \phi_j(\eta_1, n_3)\rangle, \langle \emptyset, \phi_j(\eta_2, n_4)\rangle)\rangle. \tag{5}$$

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 for applying the second rewriting rule.

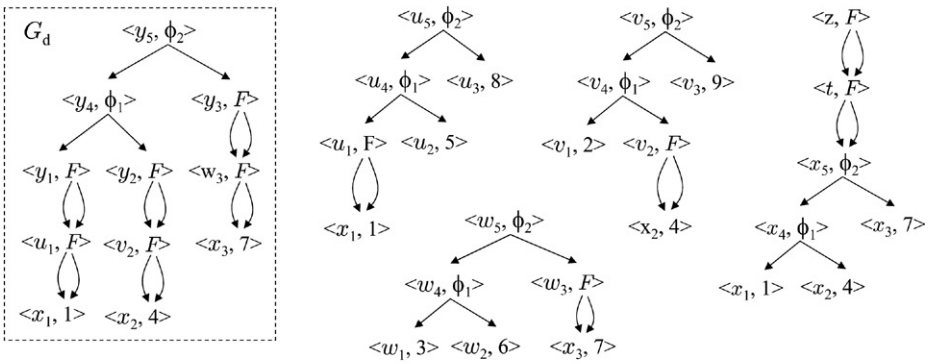
$$P : \forall \text{ nodes } \eta \in \mathit{succ}^*(\{\langle V_1, F(\eta_1, \eta_2)\rangle, \langle V_2, F(\eta_3, \eta_4)\rangle\}), \mathit{Vars}(\eta) \neq \emptyset.$$

The RKS algorithm assumes that all assignments are of the form $x := F(y, z)$, so as to make sure that for all original nodes n in the value graph, $\mathit{Vars}(\eta) \neq \emptyset$. This precondition is necessary for 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. For example, the RKS algorithm can discover all equivalences for the program in Fig. 8. Fig. 9 shows the value graph (for the program in Fig. 8) that results after applying the RKS algorithm.

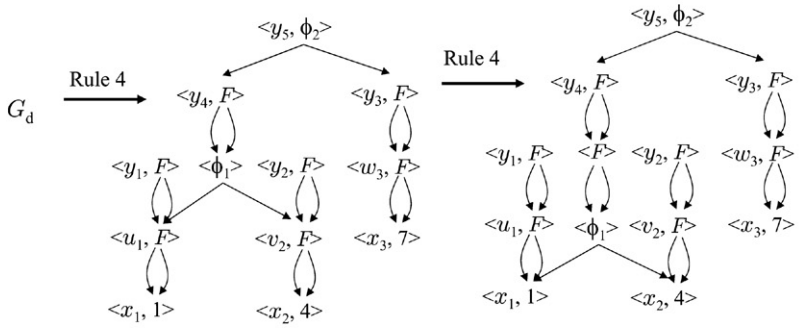
The RKS algorithm cannot discover all equivalences even in acyclic programs, contrary to what is claimed in the paper [16]. This is because the precondition P can prevent two equal expressions from reaching the same normal



(a) The RKS algorithm cannot discover $z = y_5$ in this program.



(b) The value graph representation of the program in (a) after congruence partitioning.



(c) The resultant subgraph after applying Rule 5 transformation to subgraph G_d in (b).

Fig. 10. The RKS algorithm cannot discover all equivalences even in acyclic programs.

form. Fig. 10(a) shows a program for which the RKS algorithm fails to infer the equivalence of the two program variables z and x_5. Fig. 10(b) shows the value graph representation of the program after the congruence partitioning step. Fig. 10(c) shows the value graph representation after an exhaustive application of the rewrite rules 4 and 5. The precondition P prevents any further applications of rule 5, which is necessary for merging the nodes labeled with z and y_5.

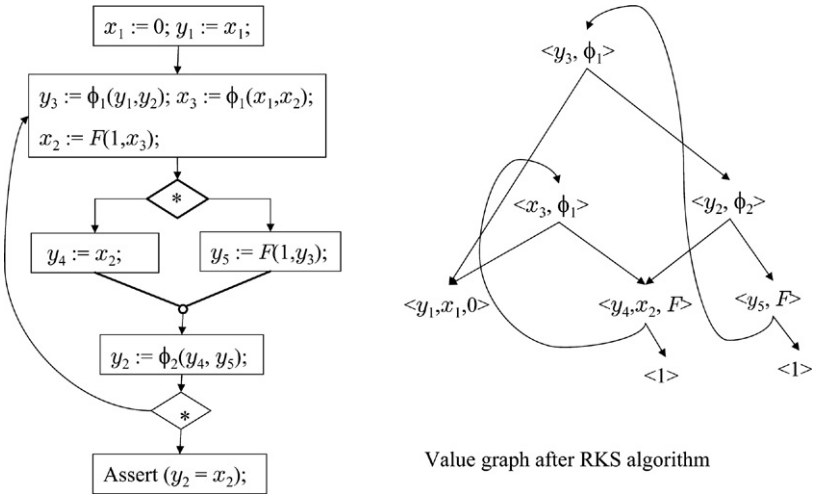


Fig. 11. The RKS algorithm cannot discover that $x_2 = y_2$ in this cyclic program even if precondition P is lifted.

On the other hand, lifting the 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. Fig. 11 shows an example where the RKS algorithm fails to discover all equivalences even if the precondition P for applying the rewrite rules is lifted. In this example, the RKS algorithm fails to discover the equality of variables x_2 and y_2 in Fig. 11 at the end of the loop. Note that detecting the equality of y_2 and x_2 requires that the ϕ_2-operator applied to y_4 and y_5 is identified as an unnecessary one (by Rule 4). This cannot be achieved, however, since it would require pre-knowledge of the value equivalence of x_3 and y_3 at node m. Congruence partitioning, however, is not able to do so, because it requires the Rule 4 simplification. This cyclic dependency between Rule 4 and congruence partitioning cannot be resolved.

5.4. Other related work

Gulwani and Necula 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.

Gulwani, Tiwari and Necula recently gave a join operation for the theory of uninterpreted functions [7]. They showed that the join operations used in the AWZ algorithm, RKS algorithm, and the algorithm described in this paper can all be cast as specific instantiations of their join operation. This suggests a possibility of a more powerful abstract interpretation for the theory of uninterpreted functions using that join operation.

Müller-Olm, Seidl, and Steffen have shown that if conditionals with equality guards are taken into account, then the problem of determining whether a specific equality holds at a program point or not is undecidable [10]. They have presented an analysis of Herbrand equalities that takes disequality guards into account.

Müller-Olm, Seidl, and Steffen have given an algorithm to detect Herbrand equalities in an interprocedural setting [11]. Their algorithm is complete (i.e., it detects all valid Herbrand equalities) for side-effect-free functions. Their algorithm can also detect all Herbrand constants.

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 a time that grows linearly with s. An interesting theoretical question is to figure out if there exist representations that could avoid the exponential lower bound for representing the set of all Herbrand equivalences.

An interesting direction for 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

We thank Oliver Rüthing for sending us the example in Fig. 11 with a useful explanation.

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.

Appendix A. Proofs

A.1. Proof of Proposition 3

The proof is by induction on the sum of the height of nodes η_1 and η_2 in G_1 and G_2 respectively.

The base case corresponds to the case when $t_1 = \bot$ or $t_2 = \bot$. Without loss of generality, let us assume that $t_1 = \bot$. Hence, $t = \bot$. Let $T_1 = \{F(e_1, e_2) \mid e_1 \in \mathit{Terms}(\ell_2), e_2 \in \mathit{Terms}(r_2)\}$ if $t_2 = F(\ell_2, r_2)$, and $T_1 = \emptyset$ if $t_2 = \bot$. Thus,

$$\begin{aligned}\mathit{Terms}(\eta) &= \mathit{Terms}(\langle V, \bot\rangle) = V = V_1 \cap V_2\\ &= V_1 \cap (V_2 \cup T_1)\\ &= \mathit{Terms}(\eta_1) \cap \mathit{Terms}(\eta_2).\end{aligned}$$

For the inductive case, $t_1 = F(\ell_1, r_1)$ and $t_2 = F(\ell_2, r_2)$. Let $\ell = \texttt{Intersect}(\ell_1, \ell_2)$ and $r = \texttt{Intersect}(r_1, r_2)$.

Let $T_2 = V \cup \{F(e_1, e_2 \mid e_1 \in \mathit{Terms}(\ell), e_2 \in \mathit{Terms}(r)\}$. Note that $t = \bot$ or $t = F(\ell, r)$. If $t = \bot$, then either $\ell = \langle\emptyset, \bot\rangle$ or $r = \langle\emptyset, \bot\rangle$. Hence, $T_2 = V \cup \emptyset = V$ and thus $\mathit{Terms}(\eta) = V = T_2$. If $t = F(\ell, r)$, then clearly $\mathit{Terms}(\eta) = T_2$. Thus, in either case $\mathit{Terms}(\eta) = T_2$.

We first prove that $\mathit{Terms}(\eta) \subseteq \mathit{Terms}(\eta_1) \cap \mathit{Terms}(\eta_2)$. It follows from the inductive hypothesis on ℓ_1 and ℓ_2 that $\mathit{Terms}(\ell) \subseteq \mathit{Terms}(\ell_1) \cap \mathit{Terms}(\ell_2)$. Similarly, it follows from the inductive hypothesis on r_1 and r_2 that $\mathit{Terms}(r) \subseteq \mathit{Terms}(r_1) \cap \mathit{Terms}(r_2)$.

$$\begin{aligned}\mathit{Terms}(\eta) &= V \cup \{F(e_1, e_2) \mid e_1 \in \mathit{Terms}(\ell), e_2 \in \mathit{Terms}(r)\}\\ &\subseteq (V_1 \cap V_2) \cup \{F(e_1, e_2) \mid e_1 \in \mathit{Terms}(\ell_1) \cap \mathit{Terms}(\ell_2),\\ &\qquad\qquad e_2 \in \mathit{Terms}(r_1) \cap \mathit{Terms}(r_2)\}\\ &= (V_1 \cap V_2) \cup (\ \{F(e_1, e_2) \mid e_1 \in \mathit{Terms}(\ell_1), e_2 \in \mathit{Terms}(r_1)\}\\ &\qquad\qquad \cap \{F(e_1, e_2) \mid e_1 \in \mathit{Terms}(\ell_2), e_2 \in \mathit{Terms}(r_2)\})\\ &= (V_1 \cup \{F(e_1, e_2) \mid e_1 \in \mathit{Terms}(\ell_1), e_2 \in \mathit{Terms}(r_1)\}) \cap\\ &\quad (V_2 \cup \{F(e_1, e_2) \mid e_1 \in \mathit{Terms}(\ell_2), e_2 \in \mathit{Terms}(r_2)\})\\ &= \mathit{Terms}(\langle V_1, F(\ell_1, r_1)\rangle) \cap \mathit{Terms}(\langle V_2, F(\ell_2, r_2)\rangle)\\ &= \mathit{Terms}(\eta_1) \cap \mathit{Terms}(\eta_2).\end{aligned}$$

We now prove that $\mathit{Terms}(\eta) \supseteq \{e \mid e \in \mathit{Terms}(\eta_1) \cap \mathit{Terms}(\eta_2), \mathit{size}(e) \leq \alpha\}$. Let α_1 and α_2 be the value of the counters when $\texttt{Intersect}(\ell_1, \ell_2)$ and $\texttt{Intersect}(r_1, r_2)$ are first called respectively. It follows from the inductive hypothesis on ℓ_1 and ℓ_2 that $\mathit{Terms}(\ell) \supseteq \{e \mid e \in \mathit{Terms}(\ell_1), e \in \mathit{Terms}(\ell_2), \mathit{size}(e) \geq \alpha_1\}$. Similarly, it follows from the inductive hypothesis on r_1 and r_2 that $\mathit{Terms}(r) \supseteq \{e \mid e \in \mathit{Terms}(r_1), e \in \mathit{Terms}(r_2), \mathit{size}(e) \geq \alpha_2\}$.

Note that α_1 is either N or $\alpha - 1$. Also, α_2 is either N or $\alpha_1 - size(e_s)$, where e_s is the smallest expression such that $e_s \in Terms(\ell_1) \cap Terms(r_1)$. Hence, $\alpha_1 \geq \alpha - 1$ and $\alpha_2 \geq \alpha - 1 - size(e_s)$.

$$
\begin{aligned}
Terms(\eta) &= V \cup \{F(e_1, e_2) \mid e_1 \in Terms(\ell), e_2 \in Terms(r)\} \\
&\supseteq V \cup \{F(e_1, e_2) \mid e_1 \in Terms(\ell_1) \cap Terms(\ell_2), \\
&\qquad\qquad size(e_1) \leq \alpha_1, \\
&\qquad\qquad e_2 \in Terms(r_1) \cap Terms(r_2), \\
&\qquad\qquad size(e_2) \leq \alpha_2\} \\
&\supseteq V \cup \{F(e_1, e_2) \mid e_1 \in Terms(\ell_1) \cap Terms(\ell_2), \\
&\qquad\qquad size(e_1) \leq \alpha - 1, \\
&\qquad\qquad e_2 \in Terms(r_1) \cap Terms(r_2), \\
&\qquad\qquad size(e_2) \leq \alpha - 1 - size(e_s)\ \} \\
&\supseteq V \cup \{F(e_1, e_2) \mid e_1 \in Terms(\ell_1) \cap Terms(\ell_2), \\
&\qquad\qquad size(e_1) \leq \alpha - 1, \\
&\qquad\qquad e_2 \in Terms(r_1) \cap Terms(r_2), \\
&\qquad\qquad size(e_2) \leq \alpha - 1 - size(e_1)\ \} \\
&= V \cup \{F(e_1, e_2) \mid e_1 \in Terms(\ell_1) \cap Terms(\ell_2), \\
&\qquad\qquad e_2 \in Terms(r_1) \cap Terms(r_2), \\
&\qquad\qquad size(F(e_1, e_2)) \leq \alpha\} \\
&= V \cup \{F(e_1, e_2) \mid e_1 \in Terms(\ell_1) \cap Terms(\ell_2) \\
&\qquad\qquad e_2 \in Terms(r_1) \cap Terms(r_2), \\
&\qquad\qquad size(F(e_1, e_2)) \leq \alpha\} \\
&= V \cup \{F(e_1, e_2) \mid F(e_1, e_2) \in Terms(\eta_1) \cap Terms(\eta_2), \\
&\qquad\qquad size(F(e_1, e_2)) \leq \alpha\} \\
&= \{e \mid e \in Terms(\eta_1) \cap Terms(\eta_2), size(e) \leq \alpha\}.
\end{aligned}
$$

A.2. *Proof of Lemma 5*

The proof is by induction on the number of operations performed by the abstract interpreter. The base case is trivial, since G does not imply any non-trivial relationship. For the inductive case, the following cases arise:

- Assignment Node. See Fig. 1(a) and Fig. 3(a).
  Suppose that $e_1 = e_2$ holds after the assignment node, and $size(e_1)$ and $size(e_2)$ is at most $s' - m$. We show that $G \models e_1 = e_2$. Let $e'_1 = e_1[^{e}/_{x}]$, and $e'_2 = e_2[^{e}/_{x}]$. Note that $e'_1 = e'_2$ holds before the assignment node, and $size(e'_1)$ and $size(e'_2)$ is at most $s' - m + size(e)$. Hence, it follows from the induction hypothesis on G' that $G' \models e'_1 = e'_2$. It now follows from Lemma 1 that $G \models e_1 = e_2$.
- Non-det Assignment Node. See Fig. 1(b) and Fig. 3(b).
  The proof of this case is similar to the case for assignment node. (Consider the expressions $e'_1 = e_1[^{x'}/_{x}]$, and $e'_2 = e_2[^{x'}/_{x}]$, where x' is a fresh variable that does not occur in G'.)
- Conditional Node. See Fig. 1(c) and Fig. 3(c).
  This case is trivial.
- Join Node. See Fig. 1(d) and Fig. 3(d).
  The proof of this case follows easily from Lemma 4.

References

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

[2] C. Click, Global code motion/global value numbering, in: Proccedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, June 1995, pp. 246–257.
[3] P. Cousot, 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, 1977, pp. 234–252.
[4] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems 13 (4) (1990) 451–490.
[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, vol. 37, 5, ACM Press, June, 2002, pp. 45–56.
[6] S. Gulwani, G.C. Necula, Global value numbering using random interpretation, in: 31st Annual ACM Symposium on POPL, ACM, January 2004.
[7] S. Gulwani, A. Tiwari, G.C. Necula, Join algorithms for the theory of uninterpreted functions, in: 24th Conference on Foundations of Software Technology and Theoretical Computer Science, in: LNCS, vol. 3328, Springer-Verlag, December 2004.
[8] G.A. Kildall, A unified approach to global program optimization, in: 1st ACM Symposium on Principles of Programming Language, October 1973, pp. 194–206.
[9] S.S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, San Francisco, 2000.
[10] M. Müller-Olm, O. Rüthing, H. Seidl, Checking herbrand equalities and beyond, in: Verification Meets Model-Checking and Abstract Interpretation, in: LNCS, vol. 3385, Springer-Verlag, January 2005.
[11] M. Müller-Olm, H. Seidl, B. Steffen, Interprocedural herbrand equalities, in: Proceedings of the European Symposium on Programming, in: LNCS, Springer-Verlag, 2005.
[12] G.C. Necula, Translation validation for an optimizing compiler, in: Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation, ACM SIGPLAN, June 2000, pp. 83–94.
[13] A. Pnueli, M. Siegel, E. Singerman, Translation validation, in: B. Steffen (Ed.), Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, in: LNCS, vol. 1384, Springer, 1998, pp. 151–166.
[14] B.K. Rosen, M.N. Wegman, F.K. Zadeck, Global value numbers and redundant computations, in: 15th Annual ACM Symposium on Principles of Programming Languages, ACM, 1988, pp. 12–27.
[15] O. Rüthing, J. Knoop, B. Steffen, The value flow graph: A program representation for optimal program transformations, in: N.D. Jones (Ed.), Proceedings of the European Symposium on Programming, in: LNCS, vol. 432, Springer-Verlag, 1990, pp. 389–405.
[16] O. Rüthing, J. Knoop, B. Steffen, Detecting equalities of variables: Combining efficiency with precision, in: Static Analysis Symposium, in: Lecture Notes in Computer Science, vol. 1694, Springer, 1999, pp. 232–247.
[17] M.N. Wegman, F.K. Zadeck, Constant propagation with conditional branches, ACM Transactions on Programming Languages and Systems 13 (2) (1991) 181–210.