Detecting Equalities of Variables: Combining Efficiency with Precision

Oliver Rüthing, Jens Knoop, and Bernhard Steffen

Department of Computer Science, University of Dortmund, Germany
{ruething,knoop,steffen}@ls5.cs.uni-dortmund.de

Abstract. Detecting whether different variables have the same value at a program point is generally undecidable. Though the subclass of equalities, whose validity holds independently from the interpretation of operators (Herbrand-equivalences), is decidable, the technique which is most widely implemented in compilers, value numbering, is restricted to basic blocks. Basically, there are two groups of algorithms aiming at globalizations of value numbering: first, a group of algorithms based on the algorithm of Kildall, which uses data flow analysis to gather information on value equalities. These algorithms are complete in detecting Herbrand-equivalences, however, expensive in terms of computational complexity. Second, a group of algorithms influenced by the algorithm of Alpern, Wegman and Zadeck. They do not fully interpret the control flow, which allows them to be particularly efficient, however, at the price of being significantly less precise than their Kildall-like counterparts. In this article we discuss how to combine the best features of both groups by aiming at a fair balance between computational complexity and precision. We propose an algorithm, which extends the one of Alpern, Wegman and Zadeck. The new algorithm is polynomial and, in practice, expected to be almost as efficient as the original one. Moreover, for acyclic control flow it is as precise as Kildall's one, i. e. it detects all Herbrand-equivalences.

1 Motivation

Detecting whether different variables have the same value at a program point is of major importance for program optimization, since equality information is a prerequisite of a broad variety of optimizations like *common subexpression elimination*, *register allocation* [10], *movement of invariant code* [13,16], branch elimination and branch fusion [10]. An even more comprehensive list is given in [1].

Unfortunately, the *equality problem*, i. e. the problem of determining whether two variables have the same value at a program point is generally undecidable. This holds even if control-flow branches are fully nondeterministically treated [12]. On the other hand, the equality problem is decidable for the subclass of equalities, whose validity holds independently from the interpretation of operators (Herbrand-equivalences[1]). In practice, however, the equality problem is

[1] In [13] Herbrand-equivalence is called *transparent equivalence*.

A. Cortesi, G. Filé (Eds.): SAS'99, LNCS 1694, pp. 232–247, 1999.

usually only tackled on program fragments like (extended) basic blocks. The state-of-the-art is here characterized by the technique of *value numbering* [4], which is implemented in many experimental and production compilers.

Basically, there are two groups of algorithms aiming at globalizations of value numbering. First, there is a group of algorithms focusing on precision. They are majorly influenced by Kildall's pioneering work [8], which is based on data flow analysis.[2] Kildall's method decides the equality problem of variables for the class of Herbrand-equivalences. Its power, however, has its price in terms of computational complexity. This is probably one of the major obstacles opposing to its widespread usage in program optimization. In [16,9] a variant of this algorithm is used in the context of semantic code motion.

Second, there is a group of algorithms paying more attention to efficiency. Typical examples are the algorithms of Alpern, Wegman and Zadeck [1] or its precursors of Reif and Lewis [12]. Also the algorithm of Fong, Kam and Ullman [6], whose results are less precise than those of the two mentioned before, falls into this group. Characteristic for approaches of this group is a more restricted treatment of the control flow of the program. In contrast to the algorithms of the first group, where the control flow is fully interpreted, the branching structure is treated to a large extent in a "syntactic" fashion. As a consequence, they are significantly less precise, but, on the other hand, surprisingly efficient, i. e. almost of linear time complexity. Like Kildall's algorithm also the algorithm of Alpern, Wegman and Zadeck has been used in the context of semantic code motion [3,2,14].

In this article we are going to show how to combine the best of both worlds. Our approach is based on the algorithm of Alpern, Wegman and Zadeck. It extends their approach by a normalization process, which resolves anomalies caused by the syntactic treatment of the control flow in the original algorithm. Our algorithm is of polynomial worst-case time complexity and, in practice, expected to be almost as efficient as the original one while, for acyclic control flow, being as precise as Kildall's one, i. e. it is complete for the class of Herbrand-equivalences. We conjecture that our result can be extended to arbitrary control flow. This would provide the first polynomial time algorithm for the detection of all Herbrand-equivalences.[3] For the sake of presentation, but without loss of generality, we restrict ourselves in this article to the equality problem of variables in a program. However, all approaches considered can easily be extended to the equality problem of expressions.

The article is organized as follows. After introducing some basic notations and definitions in Section 2, we briefly recall the two major alternate approaches for detecting global value equalities in Section 3 and 4. Central is then Section 5, where we present our extension to the partitioning approach of Alpern, Wegman,

[2] In [15] Steffen shows how Kildall's approach can be embedded into the framework of abstract interpretation with respect to the Herbrand-semantics.

[3] To our knowledge, the best known worst-case complexity estimation of advanced Kildall-like algorithms is exponential. The estimation given in [8] is here misleading.

and Zadeck. Finally, we present our conclusions and a discussion on future work in Section 6.

2 Preliminaries

We consider procedures of imperative programs, which we represent by means of directed *flow graphs* $G = (N, E, \mathbf{s}, \mathbf{e})$ with node set N, edge set E, a unique *start node* $\mathbf{s}$ and *end node* $\mathbf{e}$, which are assumed to have no incoming and outgoing edges, respectively. The nodes of G represent the statements and the edges the nondeterministic control flow of the underlying procedure. We assume that all statements are either the *empty statement* "skip" or *3-address assignments* of the form $x := y$ or $x := y_1 \ \omega \ y_2$ where x, y, y_1, y_2 are variables and ω a binary operator. By $\mathbf{P}[m, n]$ we denote the set of all finite paths from m to n. Without loss of generality we assume that every node $n \in N$ lies on a path from $\mathbf{s}$ to $\mathbf{e}$. A node m is *dominated* by a node n, if every path leading from $\mathbf{s}$ to m contains an occurrence of n. A node n dominating m with $n \neq m$ is a *strict dominator* of m, and a strict dominator of m that is dominated by all other other dominators of m is an *immediate dominator*.

The *semantics* of terms, which as usual are inductively composed of variables, constants, and operators, is considered with respect to the *Herbrand interpretation* $\mathcal{H} = (\mathbf{T}, \mathcal{H}_0)$. Here, $\mathbf{T}$ denotes the data domain given by the set of terms, and $\mathcal{H}_0$ the interpretation function, which maps every constant c to c and every operator ω to the total function $\mathcal{H}_0(\omega) : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ defined by $\mathcal{H}_0(\omega)(t_1, t_2) \stackrel{\text{def}}{=} t_1 \omega t_2$. Denoting the set of all *Herbrand states* by $\Sigma = \{\sigma \mid \sigma : \mathbf{V} \to \mathbf{T}\}$ and the distinct *start state*, which is the identity on $\mathbf{V}$, by σ_0, the *semantics* of terms $t \in \mathbf{T}$ is given by the *Herbrand semantics* $\mathcal{H} : \mathbf{T} \to (\Sigma \to \mathbf{T})$. It is inductively defined by:

$$\mathcal{H}(t)(\sigma) \stackrel{\text{def}}{=} \begin{cases} \sigma(v) & \text{if } t = v \text{ is a variable} \\ \mathcal{H}_0(c) & \text{if } t = c \text{ is a constant} \\ \mathcal{H}_0(\omega)(\mathcal{H}(t_1)(\sigma), \mathcal{H}(t_2)(\sigma)) & \text{if } t = t_1 \, \omega \, t_2 \end{cases}$$

Every node of a flow graph is associated with a state transformation and a backward-substitution function. If $n \equiv x := t$ the corresponding state transformation is defined by $\theta_n(\sigma) \stackrel{\text{def}}{=} \sigma[\mathcal{H}(t)(\sigma)/x]$, and the backward-substitution of n for a term t' is defined by $\delta_n(t') \stackrel{\text{def}}{=} t'[t/x]$. If n equals skip, both functions are the identity on their domain. Both θ_n and δ_n can naturally be extended to finite paths. This allows the following definition. Two expressions t_1 and t_2 are called *Herbrand-equivalent* at the exit of node n iff[4]

$$\forall p \in \mathbf{P}[\mathbf{s}, n]. \ \mathcal{H}(t_1)(\theta_p(\sigma_0)) = \mathcal{H}(t_2)(\theta_p(\sigma_0))$$

[4] Entry equivalence can be defined analogously.

3 Kildall-Like Algorithms at a Glance

Kildall's algorithm [8] uses data flow analysis for tackling the equality problem of variables in a program. In his original proposal equalities are represented as finite structured partitions. Rather than going into the technical details we illustrate the principal ideas by means of two meaningful examples, which are shown in Figure 1 and in Figure 2.

In the program of Figure 1 the variables x and y have the same value at the exit of node **4**.

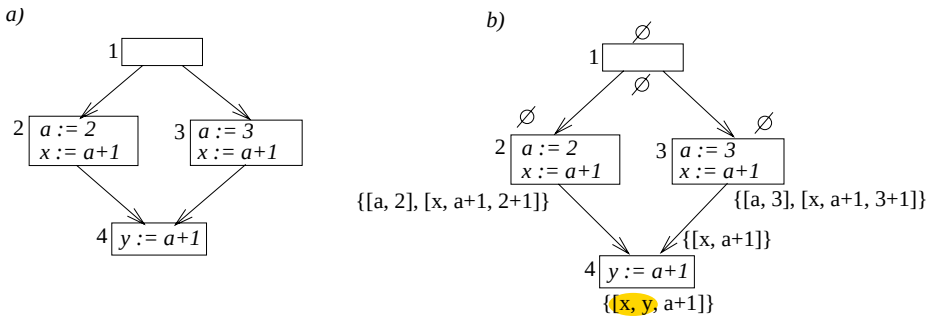


Fig. 1. Illustrating Kildall's algorithm: (a) the original program and (b) the program annotated with structured partitions.

Kildall's algorithm computes for every node entry and node exit in the program a structured partition, which characterizes all Herbrand-equivalences involving a variable. Formally, a (finite) structured partition π is a partition,[5] which

1. comprises the set of variables and expressions occurring in the program and
2. and satisfies the following consistency constraint:

$$(e, e_1 \,\omega\, e_2) \in \pi \;\wedge\; (e_1, e_1') \in \pi \;\wedge\; (e_2, e_2') \in \pi \;\Rightarrow\; (e, e_1' \,\omega\, e_2') \in \pi$$

An assignment at a node $n \equiv x := t$ is associated with a *local flow function* f_n defined by

$$f_n(\pi) \stackrel{\texttt{def}}{=} \{(t_1, t_2) \mid (\delta_n(t_1), \delta_n(t_2)) \in \pi\}.$$

The meet of two structured partitions π_1 and π_2 is given by their intersection.

Actually, it is sufficient to represent only those classes containing a variable and at least one additional element. This "sparse" representation can inductively be extended to cover an arbitrary large universe of expressions. An algorithm constructing such a minimal representation has been proposed in [16].

[5] Partitions can alternatively be considered equivalence relations on expressions. This view is exploited in the following definitions.

In order to cope with loops the annotation is computed as the greatest fixed point of an iteration sequence. It starts with the optimistic assumption that all expressions are equal (universal data flow information $\top$) except for the start node where the choice of the empty partition reflects that no value equalities are known at the entry of the program (see Figure 2 for illustration).

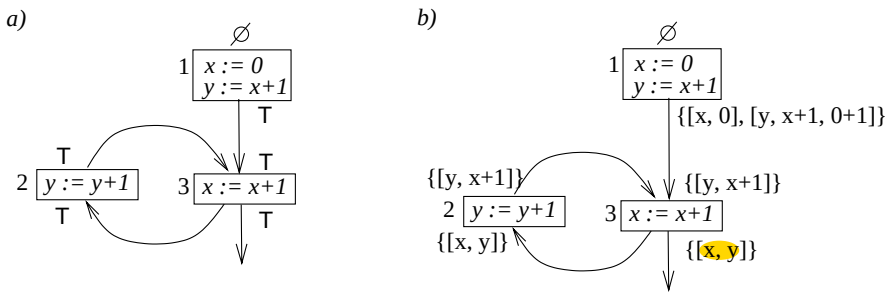


Fig. 2. Treatment of loops by Kildall's algorithm: (a) program with optimistic partition annotation and (b) the greatest fixed point annotation reveals the equality of x and y at the exit of node **3**.

3.1 Results

Kildall's algorithm is precise for the class of Herbrand-equivalences. We have the following soundness and completeness result [15]:

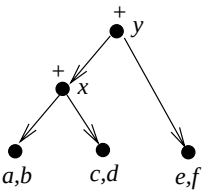
Theorem 1 (Soundness and Completeness). *Two program variables x and y are Herbrand-equivalent at a program point n if and only if (x, y) is contained in the partition annotating n after termination of Kildall's algorithm.*

Unfortunately, the precision of Kildall's algorithm has its price in terms of computational complexity. In its original formulation the growth of the size of partition classes is exponential in the number of classes in the partition. The following structured partition makes this behaviour evident:

$$\pi_{\mathtt{exp}} \stackrel{\mathtt{def}}{=} \{[a,b],[c,d],[e,f],[x,a+c,a+d,b+c,b+d],\\ [y,x+e,x+f,(a+c)+e,\ldots,(b+d)+f]\}$$

Besides the size of the partition classes also their number is problematic. Obviously, the meet of two partitions π_1 and π_2 is in the worst case of order $\Omega(|\pi_1|\,|\pi_2|)$, where $|\pi_i|$ $(i = 1, 2)$ refers to the number of classes in π_i, respectively. Unfortunately, even on an acyclic program path p, a partition might be subjected to a number of meet operations of order $\Omega(|p|)$. Together, a naive estimation yields an exponential growth of the number of classes of partitions (though no program is known exhibiting this behaviour).

In [16] we proposed a representation of structured partitions in terms of *structured partition DAGs*, in which common substructures are shared [6]. This dramatically reduces the space requirements as can be seen by the structured partition DAG to the right. It represents the partition $\pi_{\texttt{exp}}$ considered above.



The usage of structured partition DAGs eliminates the exponential blow-up in the representation of partition classes sketched above. However, the problem of the growth of the number of classes is still present. Though as mentioned above there is no example known exhibiting this exponential behaviour, one is still faced with a quite extensive data structure where every program point is annotated with a possibly large structured partition DAG. We suspect that this is the main obstacle for the widespread usage of Kildall-like techniques.

4 Alpern, Wegman and Zadeck's Algorithm at a Glance

Similar to the previous section we focus on the essential steps and ideas underlying Alpern, Wegman and Zadeck's algorithm, or for short AWZ-algorithm. We illustrate its essence on an informal and intuitive level. For this purpose we consider the example of Figure 3(a) which is a slight variant of Figure 2(a).[6] As mentioned in Section 1, the AWZ-algorithm works on flow graphs in *static single assignment* (SSA) form [5]. In essence, this means that the variables of the original program are replaced by new versions such that every variable has a unique initialization point. At merge points of the control flow pseudo-assignments $x_k := \phi_n(x_{i1}, \ldots, x_{ik})$ are introduced meaning that x_k gets the value of x_{ij} if the join node is entered via the jth ingoing edge.[7] The SSA form of our running example is depicted in Figure 3(b).

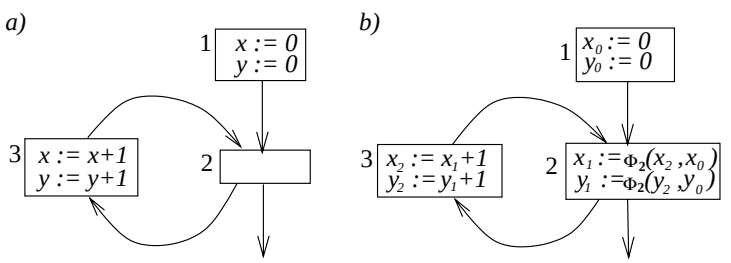


Fig. 3. Illustrating the AWZ-algorithm: (a) the original program and (b) the program transformed into SSA form.

Based on the SSA form of a program the *value graph* is constructed. It represents the value transfer of SSA variables along the control flow of the program.

[6] Actually, the AWZ-algorithm fails on the example of Figure 2.

[7] ϕ-operators are indexed by their corresponding join node.

Following the description in [10] the value graph is defined as a labelled directed graph where

- the nodes correspond to occurrences of nontrivial assignments, i. e. assignments whose right-hand side contain at least one operator, and to occurrences of constants in the program. Every node is labelled with the corresponding constant or operator symbol. Additionally, every node is annotated by the set of variables whose value is generated by the corresponding constant or assignment. An operator node is always annotated with the left-hand side variable of its corresponding assignment. Moreover, for a trivial assignment $x := y$ the generating assignment of x is defined as the generating assignment of y, and for a trivial assignment $x := c$ the corresponding node associated with c is annotated with x. For convenience, the constant or operator label is drawn inside the circle visualizing the node, and the variable annotation outside.
- Directed edges point to the operands of the right-hand side expression associated with the node. Moreover, edges are labelled with natural numbers according to the position of operands.[8]

Figure 4(a) shows the value graph corresponding to Figure 3(b). It is worth noting that the value graph is cyclic which is due to self-dependencies of variables in the loop.

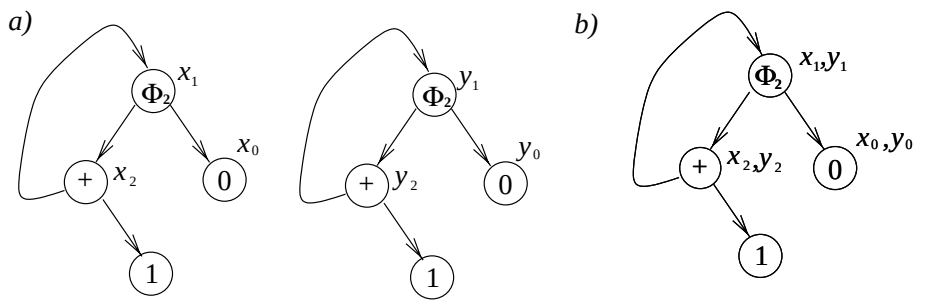


Fig. 4. (a) The value graph corresponding to Figure 3(b), and (b) the collapsed value graph of (a) after congruence partitioning.

The central step of the AWZ-algorithm is a partitioning procedure determining congruent nodes in the value graph. Like Kildall's algorithm the AWZ-algorithm proceeds optimistically computing a greatest fixed point. To this end, it starts with a coarse partition which is refined in the sequel. More precisely, the schedule of the algorithm is as follows:

Start partition: all nodes of the value graph with identical constant or operator label are grouped into the same class of the partition.

[8] We omit this labelling in our examples making the implicit assumption that edges are ordered from left to right.

Refining a partition: two nodes n and m, which belong to the same class and are labelled with a $k-$ary operator are separated into different classes if there is an $i \leqslant k$ such that the ith operands of both nodes belong to different classes.

For our illustrating example the AWZ-algorithm comes up with the *collapsed value graph* depicted in Figure 4(b).

4.1 Results

As shown in [1] the AWZ-algorithm is sound. In terms of the original flow graph the result of Alpern, Wegman and Zadeck would read as follows:

Theorem 2 (Soundness). *Two variables are Herbrand-equivalent at a program point n if there is a node in the collapsed value graph, which is annotated by the current SSA-instances of both variables.*

Here, the notion of a current SSA-instance refers to the SSA-version of the variable, which immediately dominates the program point. Unfortunately, the "only-if" direction does not hold. This means that the AWZ-algorithm is not complete (cf. Section 4.2). However, it can very efficiently be implemented by means of a technique, which resembles Hopcroft's algorithm for the minimization of finite automata [7]. For a value graph with $\mathtt{e}$ edges the AWZ-algorithm terminates within $\mathcal{O}(\mathtt{e}\ \log(\mathtt{e}))$ steps. In contrast to Kildall-like approaches, which we discussed in Section 3, it is pragmatically advantageous that the AWZ-algorithm relies on a single global data structure only, which uniformly captures both the control and the value flow.

4.2 Limitations

In this section we discuss limitations of the AWZ-algorithm and show that it is not complete. To this end we discuss typical situations, in which it fails to detect equalities of variables. Of course, according to Theorem 1 all these equalities are detected by Kildall's algorithm. The main weakness of the AWZ-algorithm is a consequence of treating ϕ-operators like ordinary operators. This way, part of the control flow is not fully interpreted, but treated in a "syntactical" way.

We elucidate this by means of the example in Figure 1(a). In this example the AWZ-algorithm fails to detect the Herbrand-equivalence of x and y at the exit of node **4**. Figure 5(a) shows the SSA form of this program and Figure 5(b) the collapsed value graph after congruence partitioning.

The reason of this failure, i.e., the failure of detecting the equality of x_2 and y_0, is that the partitioning process treats ϕ-operators like ordinary operators. Hence, even in the start partition an expression with, let's say, top-level operator "+" is separated from one with top-level operator ϕ. In other words, the AWZ-algorithm is highly sensitive to the position of ϕ-operators in composite expressions. In Section 5 we will present a normalizing transformation remedying this drawback, which is the key to our approach.

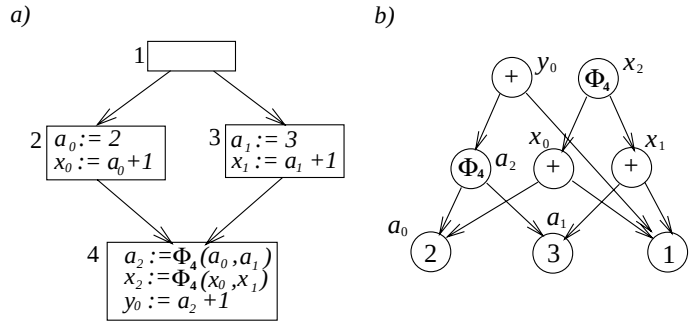


Fig. 5. Concerning Figure 1(a): the value equality of x and y at the exit of node **4** is not detected by the AWZ-algorithm. (a) The program in SSA form and (b) the collapsed value graph after congruence partitioning.

Next, we consider the looping example of Figure 2(a). Also in this example the AWZ-algorithm fails to detect the equality of x and y at the exit of node **2**.

Again the reason lies in the distinct positions of ϕ-operators, however, now in a cyclic context of the value graph. Figure 6 shows the program in SSA form and the corresponding collapsed value graph.

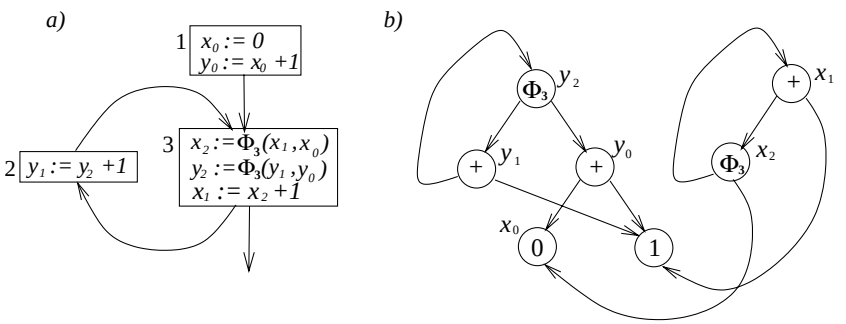


Fig. 6. Concerning Figure 2(a): the AWZ-algorithm fails to detect the value equivalence of x and y at the exit of node **4**. (a) The program in SSA form and (b) the collapsed value graph after congruence partitioning.

5 The AWZ-Algorithm with Integrated Normalization

In this section we will present our algorithm, which is an extension of the AWZ-algorithm. It works by modifying collapsed value graphs according to a set of graph rewrite rules. In order to give a precise formulation we introduce some notations and conventions.

For a node n of a collapsed value graph we denote the set of variables annotating n by $vars(n)$, its immediate successor nodes by $succ(n)$ and the set of

indirect successors by $succ^*(n)$.[9] A node n with $vars(n) = \emptyset$ is called an *anonymous* node. For the sake of presentation we assume that ϕ-operators are always binary. Note that this can always be achieved by a "linearization" of join nodes, which have more than two incoming edges. The construction of this section, however, is not tied to this assumption and can easily be extended to capture k-ary ϕ-operators as well.

5.1 The Normalization Rules

The normalization process is driven by the two graph rewrite rules depicted in Figure 7. In these rules the left-hand side of the large arrow denotes a pattern occurring in the collapsed value graph, which is replaced by the graph pattern on the right-hand side. Incoming and outgoing edges of nodes in the argument pattern which are not part of the pattern are not touched by the applications of the rules. It should be noted that separate nodes of the pattern may match the same node in the collapsed value graph. Operator labels in the pattern are matched according to the following convention: labels ϕ_n, ω in the pattern match with ϕ-operators and ordinary operators, respectively. Unlabelled nodes of the pattern match any node in the collapsed value graph.

Rule (1) is already mentioned in [1], however, only as a one-step postprocess for simplifying the value graph after termination of the congruence partitioning. In our approach Rule (1) is an integral part of the iteration process. It eliminates unnecessary ϕ-operators which can either occur as the result of the partitioning process or of applications of Rule (2). Rule (1) is applicable whenever a node n with ϕ-operator is present whose operands refer to the same node. In this case any edge pointing to n is redirected to m, the variable annotations of n are added to m, and finally, node n is eliminated.

Rule (2) is a new normalization rule. Essentially, it can be regarded as a directed distributivity rule. "Expressions" with ϕ-operators are rewritten to have ϕ-operators innermost whenever this is possible. More laxly, this rule reads as:

$$\phi_{\mathbf{m}}(a\ \omega\ b, c\ \omega\ d)\ \Rightarrow\ \phi_{\mathbf{m}}(a, c)\ \omega\ \phi_{\mathbf{m}}(b, d)$$

Rule (2) is applicable, if there is a node n with a ϕ-operator whose both operands have the same ordinary operator label, say ω, at top-level. Moreover, n must not be strictly followed by an anonymous node. The pattern is then modified as displayed on the right side of the arrow:

- Two new nodes labelled with the ϕ-operator of n are introduced and connected with the operands of l and r as depicted in Figure 7.
- Node n gets ω as its operator label.
- Finally, the outgoing edges of n are redirected to the new nodes.

Proposing a rule system directly raises questions on termination and confluence of the rewrite process where we consider congruence partitioning a graph rewriting step, too. Fortunately, both properties are satisfied.

[9] Formally, $succ^*(n)$ is the smallest set with $succ(n) \subseteq succ^*(n)$ and $\forall\, m \in succ^*(n).\ succ(m) \subseteq succ^*(n)$.

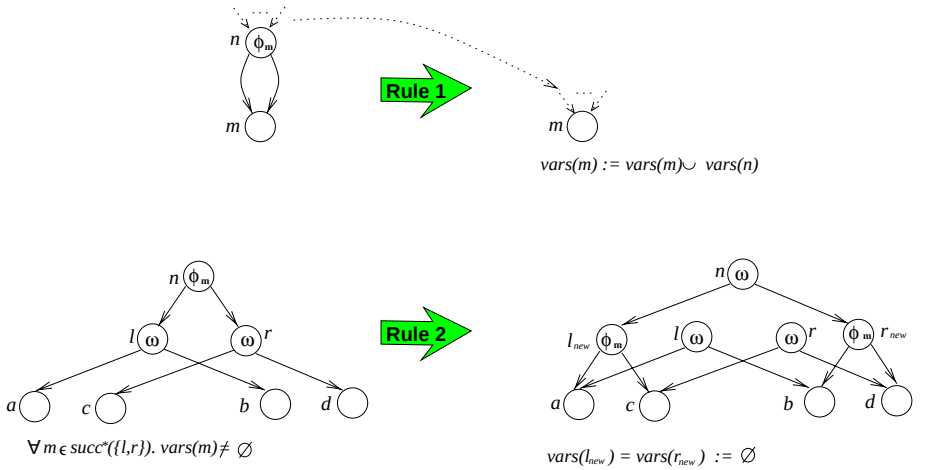


Fig. 7. Graph rewrite rules for normalizing collapsed value graphs.

Lemma 1 (Confluence and Termination). *The graph rewrite system consisting of the rules in Figure 7 together with congruence partitioning is terminating and confluent.*

Proof (Sketch): Termination is proved in the complexity section of Section 13, where the complexity of the algorithm is estimated. Thus, we concentrate here on the proof of confluence. According to Newman's Theorem [11] it suffices to prove local confluence. Obviously, congruence partitioning preserves the potential of applications of Rule (1) and Rule (2). A rule which can be applied before collapsing can also be applied after collapsing. To gain local confluence the rule has then further to be applied on all parts which are merged into the common structure. Moreover, it is easy to see that two possible applications of either Rule (1) or Rule (2) can be performed in any order. Thus, the only interesting case where two rules may overlap is a conflict between applications of Rule (1) and Rule (2). The diagram resulting from this situation together with the way of how it can be completed is shown Figure 8. □

5.2 The Iteration Strategy

The rules heavily interact which each other. For instance, Rule (1) may eliminate a ϕ-node above an operator node. This enables Rule (2). Vice versa Rule (2) may enable Rule (1) as already seen in Figure 8. In addition, Rule (1) and Rule (2) may open further opportunities for the partitioning algorithm, and vice versa it may trigger further rule applications.

In order to fully capture the interdependencies we thus propose the following schedule of the application order:

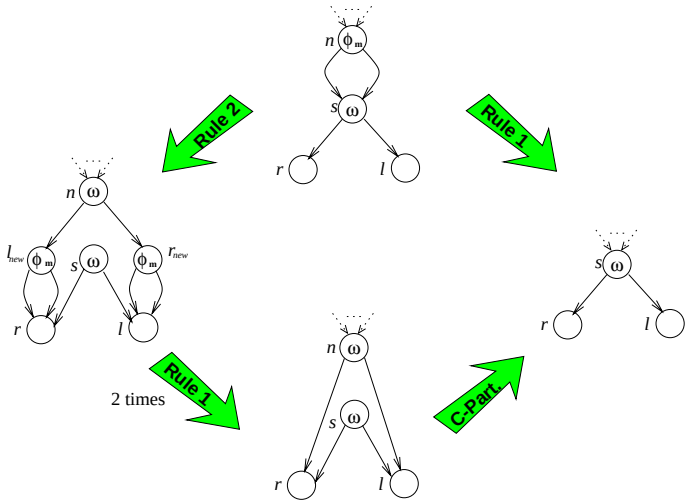


Fig. 8. Local confluence of the rewrite rules.

BEGIN
Start with the value graph as collapsed value graph.
REPEAT
1. Perform the partitioning step of the AWZ-algorithm on the collapsed value graph.
2. Apply Rule (1) and Rule (2) exhaustively.[10]

UNTIL the collapsed value graph is stable
END

The power of these rules are demonstrated by the examples of Section 4.2 which are out of the scope of the classical AWZ-algorithm. Starting with the collapsed value graph of Figure 5(b), application of Rule (2) followed by Rule (1) results in the value graph of Figure 9(a). A successive partitioning step leads to the value graph of Figure 9(b), where the equality of x and y is revealed as desired.

Also for the cyclic situation of Figure 6(b) our approach succeeds. Figure 10(a) depicts the value graph after an application of Rule (2) and Rule (1). Starting with this situation the partitioning algorithm detects the equality of x_1 and y_2 as shown in Figure 10(b).

5.3 Results

Together with the soundness of the congruence partitioning, which has been proved in [1], and the obvious soundness of Rule (1) and Rule (2) we have:

[10] After application of Rule (2) the created ϕ-nodes have to be immediately checked for applicability of Rule (1).

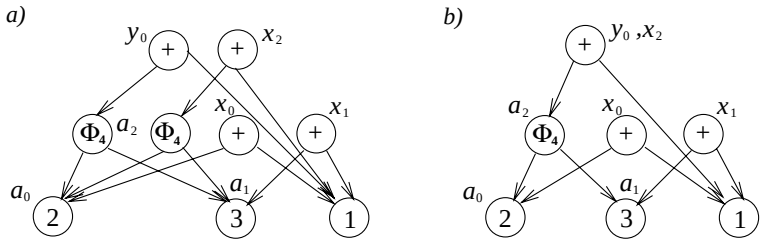


Fig. 9. The algorithm succeeds on the example of Figure 5. The collapsed value graph after (a) the application of normalization rules and (b) after a further partitioning step.

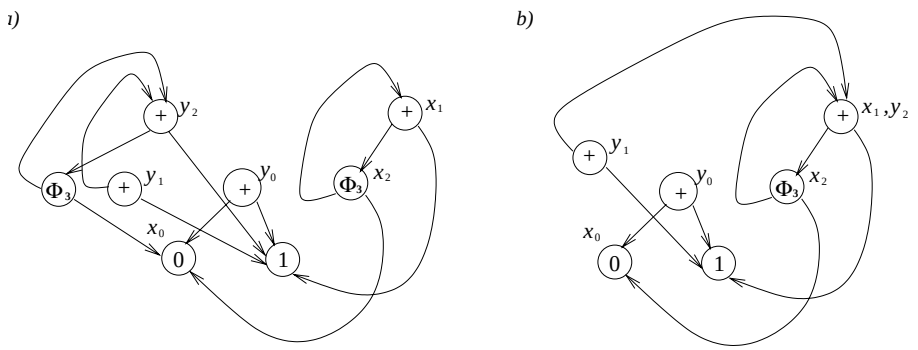


Fig. 10. The algorithm succeeds on the example of Figure 6. The collapsed value graph after (a) the application of normalization rules and (b) after a further partitioning step.

Theorem 3 (Soundness). *Two variables are Herbrand-equivalent at a program point n if there is a node in the collapsed value graph which is annotated by the current SSA-instances of both variables.*

For acyclic programs our algorithm is even complete, i.e., it detects all Herbrand-equalities. We have:

Theorem 4 (Completeness (Acyclic Case)). *In an acyclic program two program variables are Herbrand-equivalent at a program point n if and only if there is a node in the collapsed value graph which is annotated by the current SSA-instances of both variables.*

Proof (Sketch): The if-direction holds because of Theorem 3. The only-if direction, and hence completeness, can be shown by an induction on the structure of collapsed value graphs (DAGs in this case).

Let us consider two nodes n and m of a collapsed value graph[11] denoting Herbrand-equivalent terms such that n and m are not strictly followed by an anonymous node. Then we are going to show by induction on the sum of the depths of m and n that the two nodes are collapsed into a single one by our

[11] Possibly at some arbitrary intermediate stage of the transformation.

algorithm.[12] The induction base where n and m are labelled with constants is trivial. Obviously, Herbrand-equivalence forces that n and m are labelled with the same constant and thus are collapsed by the congruence partitioning step.

For the induction step we may assume that n and m refer to Herbrand-equivalent terms $op_n(u_1, u_2)$ and and $op_m(v_1, v_2)$.[13] Here we have to distinguish three different situations:

Case 1: If op_n and op_m are ordinary operators the definition of Herbrand-equivalence implies that $op_n = op_m$, say ω, and u_1 is Herbrand-equivalent with v_1 and u_2 with v_2. By the induction hypothesis the nodes belonging to u_1 and v_1 and to u_2 and v_2, respectively, are collapsed. Hence n and m are finally collapsed by congruence partitioning.

Case 2: If op_n and op_m are both ϕ-operators the situation is analogously to Case 1, if they are identical. Otherwise, let $op_n = \phi_{\mathbf{r}}$ and $op_m = \phi_{\mathbf{s}}$. Without loss of generality we may assume that $\mathbf{r}$ strictly dominates $\mathbf{s}$.[14] Then both v_1 and v_2 are also Herbrand-equivalent with $\phi_{\mathbf{r}}(u_1, u_2)$. According to the induction hypothesis m's immediate successors can be merged into a single node m'. Thus m can be eliminated by applying Rule (1) which makes the induction hypothesis applicable to n and m'.

Case 3: If op_n is an ordinary operator ω and $op_m = \phi_{\mathbf{r}}$, then we may assume that $\mathbf{r}$ strictly dominates the definition site of $\omega(u_1, u_2)$ (otherwise the ϕ-node can be eliminated with the same reasoning as in Case 2). Moreover, one may assume that $\omega(u_1, u_2)$ is immediately dominated by $\mathbf{r}$ as the backward substitution[15] $\delta_p(\omega(u_1, u_2))$ along an arbitrary path between $\mathbf{r}$ and the definition site of $\omega(u_1, u_2)$ is also charaterized by n. In order to show that both nodes belonging to v_1 and v_2 are labelled by ω one may assume that $\delta_p(\omega(u_1, u_2))$ is further backward substituted along the ingoing braches of $\mathbf{r}$. Virtually extending the value graph such that these expressions are contained, the induction hypothesis becomes applicable yielding the desired labelling of v_1 and v_2. This makes Rule (2) applicable which turns m into a node labelled with ω, too. Moreover, another application of the induction hypothesis yields that the newly introduced ϕ-operators are collapsed with u_1 and u_2 which guarantees that no anonymous ϕ-operators are introduced. Thus this case finally reduces to a situation where the reasoning of Case 1 becomes applicable. □

[12] The depth $\mathtt{d}(n)$ of a node n is 0, if n is a leaf node, and otherwise inductively defined by $max(\mathtt{d}(l(n)), \mathtt{d}(r(n))) + 1$, where $l(n)$ and $r(n)$ refer to the left and right child node of n.

[13] Note that these expressions are contained in the collapsed value graph under consideration.

[14] Note that by our assumption that n and m are Herbrand-equivalent at a certain program point $\mathbf{r}$ must strictly dominate $\mathbf{s}$ or vice versa.

[15] The backward substitution has to be slightly modified in order to take ϕ-operators into account.

Complexity By construction, ϕ-nodes added by Rule (2) only point to nodes labelled by some variable. Thus the number of newly generated nodes is bound by $\mathtt{n}^2$, where $\mathtt{n}$ denotes the number of nodes in the original value graph. As successful applications of Rule (1) and congruence partitioning steps are guaranteed to delete nodes, the total number of successful rule applications is of order $\mathcal{O}(\mathtt{n}^2)$. With the fact that congruence partitioning is the most expensive of the rewrite steps this amounts to an overall complexity of order $\mathcal{O}(\mathtt{n}^4\, log(\mathtt{n}))$. It should be noted that this is an extreme worst-case scenario. In practice, we rather expect the collapsed value graph to be linear in size of the original value graph which would reduce the computational complexity to a reasonable bound of $\mathcal{O}(\mathtt{n}^2\, log(\mathtt{n}))$.

6 Conclusions

Detecting the equality of variables (and based thereon those of expressions) is a prerequisite of a large variety of program optimizations ranging from partial redundancy elimination over common subexpression elimination to constant propagation. Since the general problem is undecidable, it is usually considered with respect to the Herbrand interpretation. With respect to this interpretation, *flow graphs*, the representation of programs most commonly used in optimization, represent value equivalences and the value flow *locally*, i. e. statementwise: the left-hand-side variable of an assignment equals the value of its right-hand-side expression. *Value flow graphs* (cf. [16]) represent the opposite pole: value equivalences of variables and expressions and the value flow are represented *globally*, i. e. across the complete program. In this scenario *value graphs* (cf. [1]) stand between flow graphs and value flow graphs with respect to performance and precision: the equivalence and value flow information represented can efficiently be computed, however, at the price of losing precision. In this article we showed how to enhance the value-graph approach in order to arrive at an algorithm which for acyclic control flow fairly combines the efficiency of the value-graph approach with the precision of the value-flow-graph approach. The resulting algorithm is optimal for acyclic programs, i. e. it detects all value equivalences with respect to the Herbrand interpretation. We are currently exploring an extension to arbitrary control flow. In particular, we are investigating an adaption of the presented strategy, in which Rule (1) and the congruence partitioning process are merged, and Rule (2) (together with Rule (1)) is exhaustively exploited in a preprocess stage. To the best of our knowledge this would provide the first algorithm for the detection of Herbrand equivalences with proven polynomial time complexity.

In addition to the theoretical perception of what is the essence of value equivalence detection, we expect that our approach has an important impact in practice as the considerably weaker basic value-graph approach of [1] is widely used in practice.

References

1. B. Alpern, M. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conf. Record of the 15th ACM Symposium on the Principles of Programming Languages* (*POPL*), January 1988.
2. P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software- Practice and Experience*, 27(6):701–724, June 1997.
3. C. Click. Global code motion/global value numbering. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI*), volume *30*,6 of *ACM SIGPLAN Notices*, pages 246–257, La Jolla, CA, June 1995.
4. J. Cocke and J. T. Schwartz. Programming languages and their compilers. Courant Institute of Mathematical Sciences, NY, 1970.
5. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependency graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451 – 490, 1991.
6. A. Fong, J. B. Kam, and J. D. Ullman. Application of lattice algebra to loop optimization. In *Conf. Record of the 2nd ACM Symposium on the Principles of Programming Languages* (*POPL*), pages 1 – 9, Palo Alto, CA, 1975.
7. J. Hopcroft. An n log n algorithm for minimizing the states of a finite automaton. *The Theory of Machines an Computations*, pages 189 – 169, 1971.
8. G. A. Kildall. A unified approach to global program optimization. In *Conf. Record of the 1st ACM Symposium on the Principles of Programming Languages* (*POPL*), pages 194 – 206, Boston, MA, 1973.
9. J. Knoop, O. Rüthing, and B. Steffen. Code motion and code placement: Just synonyms? In *Proc. 6th European Symposium on Programming* (*ESOP*), Lecture Notes in Computer Science 1381, pages 154 – 196, Lisbon, Portugal, 1998. Springer-Verlag.
10. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.
11. M. H. A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Math.*, 43,2:223–243, 1942.
12. J. H. Reif and R. Lewis. Symbolic evaluation and the gobal value graph. In *Conf. Record of the 4th ACM Symposium on the Principles of Programming Languages* (*POPL*), pages 104 – 118, Los Angeles, CA, 1977.
13. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Record of the 15th ACM Symposium on the Principles of Programming Languages* (*POPL*), pages 12 – 27, San Diego, CA, 1988.
14. L. T. Simpson. Value-driven redundancy elimination. Technical Report TR98-308, Rice University, April 6, 1998.
15. B. Steffen. Optimal run time optimization. Proved by a new look at abstract interpretations. In *Proc. 2nd International Joint Conference on the Theory and Practice of Software Development* (*TAPSOFT*), Lecture Notes in Computer Science 249, pages 52 – 68, Pisa, Italy, 1987. Springer-Verlag.
16. B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *Proc. 3rd European Symposium on Programming* (*ESOP*), Lecture Notes in Computer Science 432, pages 389 – 405, Copenhagen, Denmark, 1990. Springer-Verlag.