Join Algorithms for the Theory of Uninterpreted Functions*

Sumit Gulwani[1], Ashish Tiwari[2], and George C. Necula[1]

[1] University of California, Berkeley, CA 94720, {gulwani,necula}@cs.berkeley.edu
[2] SRI International, Menlo Park, CA 94025, tiwari@csl.sri.com

Abstract. The join of two sets of facts, E_1 and E_2, is defined as the set of all facts that are implied independently by both E_1 and E_2. Congruence closure is a widely used representation for sets of equational facts in the theory of uninterpreted function symbols (UFS). We present an optimal join algorithm for special classes of the theory of UFS using the abstract congruence closure framework. Several known join algorithms, which work on a strict subclass, can be cast as specific instantiations of our generic procedure. We demonstrate the limitations of any approach for computing joins that is based on the use of congruence closure. We also mention some interesting open problems in this area.

1 Introduction

Computational logic is used extensively in formal modeling and analysis of systems, particularly in areas such as verification of hardware and software systems, and program analysis. A wide variety of logical theories are used for this purpose. However, even for the simplest of theories, reasoning on formulas in the presence of the conjunction $\wedge$ and disjunction $\vee$ connectives is computationally hard. Unsurprisingly, therefore, almost all practical uses of logical computation have come in the form of decision procedures that work on facts stored as *conjunctions of atomic formulas*. What happens when the application requires the computation of the logical disjunction of two such facts? *Join* algorithms provide an approximate solution by constructing a conjunction of atomic formulas that is implied by the original disjunction.

Join algorithms were first studied in the context of program analysis. Abstract interpretation [5] is a well-known static program analysis technique that can be used to automatically generate program invariants, and to verify program assertions, even in the absence of loop invariants. The program is evaluated over a lattice of abstract states, each one representing one or more concrete execution states. The lattice join operation ($\sqcup$) is used to compute the abstract state following a merge in a control-flow graph, from the abstract states before the merge

* Research of the first and third authors was supported in part by NSF grants CCR-0081588, CCR-0085949, and CCR-0326577, and gifts from Microsoft Research. Research of the second author was supported in part by NSF grant CCR-0326540 and NASA Contract NAS1-20334.

point. The join operation can be viewed as computing the intersection of the facts (or union of the models) before the merge point. The lack of a suitable join algorithm restricts the utility of several interesting theories for abstract interpretation. Nevertheless, join algorithms are known for some important theories such as linear arithmetic [10], linear inequalities [6, 3], polynomial equations [17], and the initial term algebra [8, 18].

A join algorithm for a theory $\mathcal{T}h$ takes as input two sets of atomic facts and produces the strongest set of facts that is implied independently by both the input sets of facts in $\mathcal{T}h$. For example, the join of the sets $\{a = 2, b = 3\}$ and $\{a = 1, b = 4\}$ in the theory of linear arithmetic can be represented by $\{a + b = 5\}$. The join of $\{a = x, b = f(x)\}$ and $\{a = y, b = f(y)\}$ in the theory of uninterpreted function symbols (UFS) can be represented by $\{b = f(a)\}$.

It is interesting to point out that though decision procedures for satisfiability of a conjunction of atomic formulas are well studied for a wide class of logical theories, the same is not true for join algorithms. Join algorithms appear to be much harder than the decision procedures for the same theory. While there are efficient congruence closure based decision procedures for the theory of UFS, join algorithms for this theory have been studied in this paper and independently in [19]. In the special case of the theory of initial term algebra, several join algorithms have been proposed [1, 18, 8]. All of these algorithms primarily use EDAG/value graph like data structures [15, 12].

This paper has two main technical contributions. In Section 3, we present an abstract congruence closure based algorithm that generalizes all the known join algorithms for subclasses of UFS and can compute the join for a strictly bigger subclass of the theory of UFS than these algorithms. We show that the existing algorithms are a special case of our algorithm.

In Section 4 we present some results concerning the limitations of any congruence closure based approach for obtaining join algorithms for the general theory of UFS. We show that the join of two finite sets of ground equations cannot be finitely represented (using ground equations). In special cases when it can be finitely represented, the presentation can become exponential. This partially explains the lack of any known complete join algorithms for even special classes of the UFS theory.

2 Notation

A set Σ of function symbols and constants is called a *signature*. Function symbols in Σ are denoted by f, g and constants by a, b. In the context of program analysis, these constants arise from program variables, and henceforth we refer to them as (program) variables. We use $\mathcal{T}(\Sigma)$ to refer to the set of ground terms over Σ, which are constructed using symbols only from Σ. We use the notation $ft_1 \ldots t_k$ to refer to the term $f(t_1, .., t_k)$. We also use the notation $f^i t$ to denote i applications of the unary function f on term t.

Definition 1 (Join). *Let $\mathcal{T}h$ be some (first-order) theory over a signature Σ. Let E_1 and E_2 be two sets of ground equations over Σ. The* join *of E_1 and E_2*

in theory $\mathcal{T}h$ is denoted by $E_1 \sqcup_{\mathcal{T}h} E_2$, and is defined to be (any presentation for) the set $\{s = t \mid s, t \in \mathcal{T}(\Sigma), \mathcal{T}h \models E_1 \Rightarrow s = t, \mathcal{T}h \models E_2 \Rightarrow s = t\}$.

We ignore the subscript $\mathcal{T}h$ from $\sqcup_{\mathcal{T}h}$ whenever it is clear from the context. In this paper, we mainly concern ourselves with the theory of UFS. If E is a set of equations (interpreted as a binary relation over terms, not necessarily symmetric), the notation $\rightarrow_E$ denotes the closure of E under the congruence axiom. If $\rightarrow$ is a binary relation, we use the notation $\rightarrow^*$ and $\leftrightarrow^*$ to denote the reflexive-transitive and reflexive-symmetric-transitive closure of $\rightarrow$. Note that $\leftrightarrow^*_E$ is the equational theory induced by E.

The theory of *uninterpreted function symbols* (UFS) is just the pure theory of equality, that is, there are no additional equational axioms. Treating the constants in Σ as variables, we define the theory of *initial term algebra* as the extension of UFS with the axiom that if $fs_1 \ldots s_m = gt_1 \ldots t_n$ for $m, n \geq 1$, then $f = g$, $m = n$, and $s_i = t_i$ for all i.

3 Join Algorithms for Uninterpreted Functions

We represent (the equational theory induced by) finite sets of ground equations using an "abstract congruence closure" [9, 2], which is closely related to a bottom-up tree automaton where the automaton specifies an *equivalence* on a set of terms, rather than specifying a set of accepted terms. Abstract congruence closure is reviewed in Section 3.1. The join of two abstract congruence closures is closely related to their product, which we describe in Section 3.2.

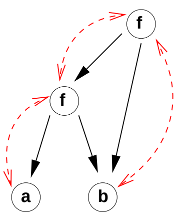
3.1 Abstract Congruence Closure

An abstract congruence closure provides a rewrite rules (tree-automata) based representation for a finite set of ground equations [2]. An *abstract congruence closure* R is a convergent set of ground rewrite rules of the form $fc_1 \ldots c_k \rightarrow c_0$ or $c_1 \rightarrow c_2$, where $f \in \Sigma$ is a k-ary function symbol ($k \geq 0$) and c_i's are all special constants from a set K disjoint from Σ. If E is a set of ground equations over Σ, then R is said to be an abstract congruence closure *for E* if for all $s, t \in \mathcal{T}(\Sigma)$, $s \leftrightarrow^*_E t$ iff there exists a term $u \in \mathcal{T}(\Sigma \cup K)$ such that $s \rightarrow^*_R u$ and $t \rightarrow^*_R u$. We will assume that R is fully-reduced, that is, if $fc_1 \ldots c_k \rightarrow c_0 \in R$, then there is no rule in R such that $c_i \rightarrow_R d$ for any $i = 0, 1, \ldots$ and any d. R is fully-reduced implies that R is convergent.

If $s \rightarrow^*_R c$, then we say that *c represents s* (via R). Given an abstract congruence closure R, it is not the case that every term $s \in \mathcal{T}(\Sigma)$ is represented by a constant. Note that an abstract congruence closure provides a formal way for reasoning about EDAG data-structure [15], as illustrated in Figure 1.

3.2 Join of Two Congruence Closures

We use congruence closures to represent sets of equations. If R_1 and R_2 are abstract congruence closures over signatures $\Sigma \cup K_1$ and $\Sigma \cup K_2$ respectively,



Consider the set $E = \{fab = a, f(fab)b = b\}$. An EDAG representing the set E is shown in Figure 1. An EDAG consists of a term graph (dark directed edges) and a set of congruence closed equality (dotted) edges. A corresponding abstract congruence closure representation is $\{a \to c_0, b \to c_1, fc_0c_1 \to c_2, fc_2c_1 \to c_3, c_1 \to c_0, c_2 \to c_0, c_3 \to c_0\}$. A fully-reduced abstract congruence closure for E is $R = \{a \to c_0, b \to c_0, fc_0c_0 \to c_0\}$. The rewrite system R can be seen as a specification of a tree-automaton over the set $K = \{c_0\}$ of *states*.

Fig. 1. EDAG and abstract congruence closure for $E = \{fab = a, f(fab)b = b\}$

then we want to construct an abstract congruence closure R_3 such that for all terms $s, t \in \mathcal{T}(\Sigma)$, it is the case that $s \leftrightarrow^*_{R_1} t$ and $s \leftrightarrow^*_{R_2} t$, if and only if, $s \leftrightarrow^*_{R_3} t$. The solution involves the construction of the *product* congruence closure.

Definition 2. *Let R_1 and R_2 be abstract congruence closures over signatures $\Sigma \cup K_1$ and $\Sigma \cup K_2$. We define the* product *congruence closure R_3 over the signature $\Sigma \cup (K_1 \times K_2)$ as follows:*

$$R_3 = \{f(\langle c_1, d_1\rangle, \langle c_2, d_2\rangle, \ldots, \langle c_k, d_k\rangle) \;\to\; \langle c, d\rangle : \\ f \in \Sigma,\; fc_1c_2 \ldots c_k \to c \in R_1,\; fd_1d_2 \ldots d_k \to d \in R_2\}$$

Example 1. Let $E_1 = \{f^2a = a\}$ and $E_2 = \{f^3a = a\}$. A fully-reduced abstract congruence closure for E_1 is $R_1 = \{a \to c_0,\; fc_0 \to c_1,\; fc_1 \to c_0\}$, and that for E_2 is $R_2 = \{a \to d_0,\; fd_0 \to d_1,\; fd_1 \to d_2,\; fd_2 \to d_0\}$.

A fully reduced abstract congruence closure for the join $E_1 \sqcup E_2$ is given over the signature $\{a, f\} \cup \{\langle c_i, d_j\rangle : i \in \{0,1\}, j \in \{0,1,2\}\}$ as $R_3 = \{a \to \langle c_0, d_0\rangle, f\langle c_0, d_0\rangle \to \langle c_1, d_1\rangle, f\langle c_1, d_1\rangle \to \langle c_0, d_2\rangle, f\langle c_0, d_2\rangle \to \langle c_1, d_0\rangle, f\langle c_1, d_0\rangle \to \langle c_0, d_1\rangle, f\langle c_0, d_1\rangle \to \langle c_1, d_2\rangle, f\langle c_1, d_2\rangle \to \langle c_0, d_0\rangle\}$. Here R_3 is just the product of R_1 and R_2.

The following lemma shows that product construction is sound (i.e., it represents only true equivalences), but complete (i.e., it represents all true equivalences) only on terms represented explicitly by constants in R_1 and R_2.

Lemma 1. *Let R_1 and R_2 be fully reduced abstract congruence closures over signatures $\Sigma \cup K_1$ and $\Sigma \cup K_2$. Let R_3 be the product congruence closure. Then, for all terms $s, t \in \mathcal{T}(\Sigma)$, it is the case that $s \to^*_{R_1} c \leftarrow^*_{R_1} t$ and $s \to^*_{R_2} d \leftarrow^*_{R_2} t$ for some constants $c \in K_1$ and $d \in K_2$, if and only if, $s \to^*_{R_3} \langle c, d\rangle \leftarrow^*_{R_3} t$.*

Proof. By induction on the structure of the term s, we can prove that it is the case that $s \to^*_{R_1} c$ and $s \to^*_{R_2} d$, iff $s \to^*_{R_3} \langle c, d\rangle$. The lemma follows immediately.

3.3 Special Cases for which the Join Algorithm is Complete

In this section, we show that for certain special cases, the product captures the exact join.

Injective Functions. An important special case, from the point of view of program analysis, is the theory of injective functions. In this case, all function symbols $f \in \Sigma$ are assumed to be injective, that is, whenever $fs_1 \ldots s_k = ft_1 \ldots t_k$, then $s_i = t_i$ for all i.

Uninterpreted functions are a commonly used abstraction for modeling program operators for the purpose of program analysis. If the conditionals of a program are abstracted as non-deterministic, and all program assignments are of the form $a := e$ (where a is a program variable and e is some uninterpreted function term), then it can be shown that if $fs_1 \ldots s_k = ft_1 \ldots t_k$ holds at some program point, then $s_i = t_i$ for all i must also be true at that program point [8]. Hence, the analysis of such programs can use the theory of injective functions. Furthermore, injective functions can be used to model fields of tree-like data structures in programs.

As a consequence of injectivity, there cannot be two *distinct* rules $f \ldots \to c$ and $f \ldots \to c$ in any fully-reduced abstract congruence closure.

Theorem 1. *Let R_1 and R_2 be fully reduced abstract congruence closures over signatures $\Sigma \cup K_1$ and $\Sigma \cup K_2$ that satisfy the injectivity assumption described above. Let R_3 be the product congruence closure. Then, the relation $\leftrightarrow^*_{R_3}$ is equal to $\leftrightarrow^*_{R_1} \cap \leftrightarrow^*_{R_2}$ over $\mathcal{T}(\Sigma)$.*

Proof. Suppose $s, t \in \mathcal{T}(\Sigma)$ such that $s \leftrightarrow^*_{R_1} t$ and $s \leftrightarrow^*_{R_2} t$. Consider the cases:
(1) *there are constants $c \in K_1$ and $d \in K_2$ equivalent to s modulo R_1 and R_2 respectively*: Since R_1 and R_2 are fully-reduced, it follows that $s \to^*_{R_1} c' \leftarrow^*_{R_1} t$ and $s \to^*_{R_2} d' \leftarrow^*_{R_2} t$ for some constants c', d'. It follows from Lemma 1 that $s \leftrightarrow^*_{R_3} t$.
(2) *there is no constant in the equivalence class of s modulo R_1*: Then $s = fs_1 \ldots s_m$ and $t = ft_1 \ldots t_m$ for some $f \in \Sigma$ and $s_i \leftrightarrow^*_{R_1} t_i$ for all i. Since s and t are also equivalent modulo R_2, by injectivity it follows that for all i, s_i and t_i are equivalent modulo R_2. By induction on the depth of s and t, we conclude that $s_i \leftrightarrow^*_{R_3} t_i$, and consequently, $s \leftrightarrow^*_{R_3} t$.
(3) *there is no constant in the equivalence class of s modulo R_2*: This case is analogous to the second case above.

Finite Number of Congruence Classes. As a second specialization, consider the case when only a finite number of distinct congruence classes are induced by both R_1 and R_2. In this case, the product of R_1 and R_2 represents the complete join of R_1 and R_2.

Theorem 2. *Let R_1 and R_2 be fully reduced abstract congruence closures over signatures $\Sigma \cup K_1$ and $\Sigma \cup K_2$. Let R_3 be the product congruence closure. If the congruence relation $\leftrightarrow^*_{R_1}$ defined by R_1 over $\mathcal{T}(\Sigma)$ induces only finitely many congruence classes, and the same is true for R_2, then the relation $\leftrightarrow^*_{R_3}$ is equal to $\leftrightarrow^*_{R_1} \cap \leftrightarrow^*_{R_2}$ over $\mathcal{T}(\Sigma)$.*

Proof. If $\mathcal{T}(\Sigma)$ is partitioned into a finite number of congruence classes modulo R_1, then we claim that every term $s \in \mathcal{T}(\Sigma)$ is equivalent to some constant modulo R_1. Thereafter the proof is identical to case (1) of the proof of Theorem 1. To prove the claim, note that if s is not equivalent to a constant, then all the infinite terms $C[s]$, where C is an arbitrary context, are in distinct equivalence classes, thus contradicting the assumption.

3.4 Complexity and Optimizations

If the size of R_1 and R_2 is n_1 and n_2 respectively, then the product R_3 of R_1 and R_2 can be constructed in $O(n_1 n_2)$ time and the size of R_3 is $O(n_1 n_2)$. Example 1 generalizes to show that in the case of finitely many equivalence classes, the abstract congruence closure representation of the (complete) join can be quadratic in size of the inputs, and hence product construction is optimal in this case. Surprisingly, the same is also true for the theory of injective functions and the special subclass of initial term algebra, as the following example demonstrates.

Example 2. Let $\Sigma = \{a_i, b_i, a'_i, b'_i \mid i \in I = \{1, \ldots, n\}\}$ be a set of $4n$ variables. It is easy to see that the join of $\{a_1 = \cdots = a_n = b_1 = \cdots = b_n = fb'_1\} \cup \{fb'_{i+1} = b_i, fa_i = a_{i+1} \mid i \in I\}$ and $\{a'_1 = \cdots = a'_n = b'_1 = \cdots = b'_n = fb_1\} \cup \{fb_{i+1} = b_i, fa_i = a_{i+1} \mid i \in I\}$ is $\{a'_i = f^i b_i, a_i = f^i b'_i \mid i \in I\}$, which can only be represented by a congruence closure of quadratic size.

In the context of program analysis, abstract interpretation of a program with n conditionals (in sequence) requires computing n successive joins. A quadratic blowup in each step can lead to a double exponential complexity. In practice, however, we would not expect the join to be quadratic in each step. Product construction can be optimized using some heuristics. First we can delete *unreachable constants*, that is, a constant $\langle c, d\rangle$ that does not represent any term over the original signature Σ. Rules that contain unreachable constants can also be deleted. This optimization can be enforced at the product construction phase by only creating constants that are guaranteed to be reachable. Second note that any node that is not pointed to by any subterm edge or any equational edge can also be recursively deleted. In other words, if $\langle c, d\rangle$ occurs exactly once, then the rule containing $\langle c, d\rangle$ can be deleted.

3.5 Related Work

We recently discovered that Vagvolgyi [19] has shown that it is decidable if the join of two congruence closures is finitely generated, and has described an algorithm for computing the join that is based heavily on tree-automata techniques. Our work has focused on identifying classes of UFS for which joins are guaranteed to be finitely generated in *polynomial* time using product construction of *abstract congruence closures*.

We next describe existing join algorithms (all of which work for a subclass of injective functions) and then show that they are specific instantiations of our join algorithm (which works for a strictly bigger class). These join algorithms have been studied in the context of the global value numbering problem. The global value numbering problem [4, 8] seeks to discover equivalences of program expressions by treating all program operators as uninterpreted, all conditionals as non-deterministic, while all assignments are of the form $a := e$ (where a is a program variable and e is some uninterpreted function term). In this special case, a congruence closure R is, in fact, a *unification closure*, that is, whenever s and t are equivalent modulo R, then (a) either s or t is a constant, or (b) $s = fs_1 \ldots s_m$, $t = fs_1 \ldots t_m$, and s_i and t_i are equivalent modulo R. The three different algorithms proposed for computing joins in the initial term algebra [1, 18, 8] can be viewed as essentially computing the product congruence closure. However, since there is a potential of computing n successive joins and getting an exponential blowup [8], these algorithms use special strategies and incomplete heuristics to optimize computation of n joins.

The popular partition refinement algorithm proposed by Alpern, Wegman, and Zadeck (AWZ) [1] is particularly efficient, however at the price of implementing an incomplete join. The novel idea in AWZ algorithm is to represent the values of variables after a join using a fresh *selection* function ϕ, similar to the functions used in the static single assignment form [7]. 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. However, the AWZ algorithm treats the ϕ functions as new uninterpreted functions. It then performs congruence partitioning and finally eliminates the ϕ functions. For example, the join of $\{x = a, y = f(a), z = a\}$ and $\{x = b, y = f(b), z = b\}$ is represented as $\{x = \phi(a, b), y = \phi(f(a), f(b)), z = \phi(a, b)\}$ and computed to be $\{x = z\}$. Note that the equality $y = f(x)$ is missing in the join. 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) [18] that alternately applies the AWZ algorithm and some rewrite rules for normalization of terms involving ϕ functions (namely $\phi(a, a) \to a$ and $\phi(f(a), f(b)) \to f(\phi(a, b))$), until the congruence classes reach a fixed point. Their algorithm discovers more equivalences than the AWZ algorithm. Recently, Gulwani and Necula [8] gave a join algorithm (GN) for the initial term algebra that takes as input a parameter s and discovers all equivalences among terms of size at most s.

The GN algorithm. In our framework, the basic strategy of the GN algorithm [8] for computing the join of two congruence closures R_1 and R_2 can be described by the recursive function *match*:

match(c,d) =
  if $\exists a : a \to c \in R_1, a \to d \in R_2$ create $a \to \langle c, d\rangle$; return;
  else if $\exists fc_1 \ldots c_k \to c \in R_1$ and $\exists fd_1 \ldots d_k \to d \in R_2$
    create $f\langle c_1, d_1\rangle \ldots \langle c_k, d_k\rangle \to \langle c, d\rangle$; $match(c_1, d_1); \ldots; match(c_k, d_k)$;
  else delete all rules created until now;

For each variable $a \in \Sigma$ such that $a \to c \in R_1$ and $a \to d \in R_2$, the function $match(c, d)$ is invoked once. Note that rules are deleted if they contain unreachable nodes. If $R_1 = \{a \to c_1, fc_2 \to c_1, b \to c_2\}$ and $R_2 = \{a \to d_1, fd_2 \to$

$d_1, b \to d_2\}$, then the GN algorithm creates the rules $\{a \to \langle c_1, d_1\rangle, f\langle c_2, d_2\rangle \to \langle c_1, d_1\rangle, b \to \langle c_2, d_2\rangle\}$ in that order.

The RKS algorithm. This algorithm [18] uses the special ϕ function to represent the join problem. The binary ϕ function corresponds to the pairing operator $\langle \_, \_ \rangle : K_1 \times K_2 \mapsto K_3$, but extended to terms $\langle \_, \_ \rangle : \mathcal{T}(\Sigma \cup K_1) \times \mathcal{T}(\Sigma \cup K_2) \mapsto \mathcal{T}(\Sigma \cup K_3)$. The process of creating the rewrite rule $f\langle c_1, d_1\rangle \ldots \langle c_k, d_k\rangle \to \langle c, d\rangle$ from the two initial rewrite rules $fc_1 \ldots c_k \to c \in R_1$ and $fd_1 \ldots d_k \to d \in R_2$ is achieved by first explicitly representing the rewrite rule $\langle fc_1 \ldots c_k, fd_1 \ldots d_k\rangle \to \langle c, d\rangle$, and then commuting the ϕ function with the f symbol to get $f\langle c_1, d_1\rangle \ldots \langle c_k, d_k\rangle \to \langle c, d\rangle$. Finally, in the base case, when we get $\langle a, a\rangle \to \langle c, d\rangle$, the second property of ϕ functions is used to simplify the left-hand side to a.

The AWZ algorithm. The AWZ algorithm [1] also uses the special ϕ function, but does not use any of the two properties of it (as described above). Consequently, it only computes a few rewrite rules of the product congruence closure and not all of them.

4 Limits of Congruence Closure based Approaches

The congruence closure representation is inherently limited in its expressiveness. It can only represent sets of equations that have a finite presentation. However, the join of two finite sets of ground equations may not have a finite presentation. For example, consider the following sets of equations E_1 and E_2.

$$E_1 = \{a = b\} \qquad E_2 = \{fa = a, fb = b, ga = gb\}$$
$$E_1 \sqcup E_2 = \{gf^n a = gf^n b \mid n \geq 0\}$$

We prove that $E_1 \sqcup E_2$ cannot be represented by a finite number of ground equations below. We first define signature of a term.

Definition 3. *Let $\equiv$ be a congruence on the set of all ground terms. Let K denote the set of all congruence classes induced by $\equiv$. The signature $Sig(t)$ of a term $t = f(t_1, \ldots, t_k)$ with respect to $\equiv$ is the term $f([t_1], \ldots, [t_k])$ over $\Sigma \cup K$, where $[t_i]$ denotes the congruence class of t_i modulo $\equiv$ and symbols in K are treated as constants.*

The following theorem gives a complete characterization of equational theories that admit finite presentations using ground equations, see also [11].

Theorem 3. *A congruence relation $\equiv$ on the set of ground terms (over Σ) can be represented by a finite set of ground equations iff there are only finitely many congruence classes that contain terms with different signatures, and each such congruence class contains terms with only finitely many different signatures.*

We will only use the forward ($\Rightarrow$) implication of this theorem, which follows immediately using either an abstract congruence closure construction of the finite set of ground equations, or analyzing the equational proofs.

Note that the two terms $gf^n a$ and $gf^n b$, for a fixed $n \geq 0$, are equal in $E_3 = E_1 \sqcup E_2$, but their arguments are not (because $E_2 \not\models a = b$.) Thus, $E_1 \sqcup E_2$ contains infinitely many congruence classes with two distinct signatures. Hence it follows from Theorem 3 that E_3 does not admit a finite presentation using ground equations. We conclude that *the congruence closure based approach cannot be used to obtain a complete join algorithm for the full theory of UFS.* In fact, this example shows that this is true for even the special class of *unary* UFS.

A set E of ground equations is said to be *cyclic* if there exists a term that is equivalent to a proper subterm of itself modulo E, otherwise it is *acyclic.* The acyclic subclass of UFS is closed under joins and guaranteed to have finite presentations. Unfortunately, the (complete) join of two sets of acyclic ground equations can be exponential in the size of the inputs. For example, consider the following sets of equations E_1 and E_2.

$$\begin{aligned}
E_1 &= \{a = b\} \\
E_2 &= \{g(b', a) = g(b', b), b' = f(a_1, .., a_n), a_1 = a'_1, \ldots, a_n = a'_n\} \\
E_1 \sqcup E_2 &= \{g(s, a) = g(s, b) \mid s \in f(t_1, .., t_n), t_i \in \{a_i, a'_i\}\} \cup \{g(b', a) = g(b', b)\}
\end{aligned}$$

The set $E_1 \sqcup E_2$ requires an exponential number of ground equations for representation. We conclude that *the congruence closure based approach cannot be used to get a polynomial time complete join algorithm for the acyclic subclass of UFS.* This remains true even when the signature is restricted to unary symbols, as the following example shows.

$$\begin{aligned}
E_1 &= \{x_0 = y_0\} \\
E_2 &= \{fx_0 = x_1, .., fx_{n-1} = x_n, gx_0 = x_1, .., gx_{n-1} = x_n, \\
&\qquad fy_0 = y_1, .., fy_{n-1} = y_n, gy_0 = y_1, .., gy_{n-1} = y_n, x_n = y_n\} \\
E_1 \sqcup E_2 &= \{sx_0 = sy_0 \mid s \in (f|g)^n\}
\end{aligned}$$

Note that the set $E_1 \sqcup E_2$ contains 2^n equations .The smallest set of ground equations representing $E_1 \sqcup E_2$ is exponentially large.

4.1 Relatively Complete Join Algorithm

We cannot hope to get a complete join algorithm using the congruence closure, or EDAG, data-structure. We can, however, get an algorithm that is complete on a given set I of *important* terms.

Definition 4 (Relatively Complete Join Algorithm). *A* relatively complete join algorithm *for a theory $\mathcal{T}h$ over a signature Σ takes as input two sets of ground equations E_1 and E_2, and a set I of terms over Σ and returns E_3 such that $\mathcal{T}h \models (E_1 \sqcup E_2) \Rightarrow E_3$ and $\mathcal{T}h \models E_3 \Rightarrow (E_1 \sqcup E_2)|_I$, where $(E_1 \sqcup E_2)|_I = \{s = t \mid s$ and t occur as sub-terms in $I, (s = t) \in E_1 \sqcup E_2\}$.*

Lemma 1 shows that the product construction method will detect exactly those equivalences which involve terms that are explicitly represented (via constants) in the two congruence closures. Hence, we can obtain relatively complete join algorithms by first representing the set I of important terms in R_1 and R_2. Define the function $\mathit{addTerm}(K, R, s)$, which takes as input a set K of constants, an abstract congruence closure R over $\Sigma \cup K$, and a term s, and returns a tuple $\langle K', R', c\rangle$ as follows:

$$
\begin{aligned}
\mathit{addTerm}(K, R, c) &= \langle K, R, c\rangle, \ \textbf{if } c \in K \\
\mathit{addTerm}(K, R, fc_1 \ldots c_k) &= \langle K, R, c\rangle, \ \textbf{if } fc_1 \ldots c_k \to c \in R \\
\mathit{addTerm}(K, R, fc_1 \ldots c_k) &= \langle K \cup \{c\}, R \cup \{fc_1 \ldots c_k \to c\}, c\rangle, \ \textbf{if } c_i \in K, c \notin K \\
\mathit{addTerm}(K, R, f \ldots s_i \ldots) &= \mathit{addTerm}(K_1, R_1, f \ldots c \ldots), \\
&\qquad \textbf{if } \langle K_1, R_1, c\rangle = \mathit{addTerm}(K, R, s_i)
\end{aligned}
$$

The function $\mathit{addTerm}(K, R, s)$ adds new rules to R, if necessary, so that the term s is explicitly represented (by a constant) in R. We extend this function to add a set I of terms by successively calling the function $\mathit{addTerm}(K, R, s)$ for each $s \in I$. The relatively complete join algorithm *relJoin* for UFS involves adding the new terms and then computing the product.

```
relJoin(R1,R2,I) =
  ⟨K1,R1⟩ := addTerm(K1,R1,I);
  ⟨K2,R2⟩ := addTerm(K2,R2,I);
  R3 := product(R1,R2); return R3;
```

For example, consider the congruence closures $R_1 = \{a \to c, b \to c\}$ and $R_2 = \{a \to d_1, b \to d_2, fd_1 \to d_3, fd_2 \to d_3, gd_1 \to d_4, gd_2 \to d_4\}$ for $\{a = b\}$ and $\{fa = fb, ga = gb\}$ respectively. The product of R_1 and R_2 will be a congruence closure for the empty set $\{\}$ of equations since fa, fb, ga and gb are not represented in R_1. If $I = \{fa\}$, then we will add, say, the rule $fc \to c'$ to R_1, and the product now represents $\{fa = fb\}$. It will still miss the equality $ga = gb$.

The correctness of the relatively complete join algorithm outlined above follows immediately from Lemma 1 and noting that *addTerm* returns a fully reduced congruence closure if the input congruence closure is fully reduced [2]. As a post-processing step, we can only keep those rules in R_3 that are used in the proof of $s \to^*_{R_3} c$ for some $s \in I$. This way the size of the output can be forced to be linear in the size of the input I.

5 Interesting Future Extensions

Join Algorithms for Other Theories: Join algorithms for the theory of commutative UFS can be used to reason about program operators like bitwise operators and floating-point arithmetic operators. However, the join algorithm for commutative UFS (*cufs*) may be more challenging than the one for UFS. For example, consider the following sets of equations E_1 and E_2.

$$E_1 = \{a = a', b = b'\} \qquad E_2 = \{a = b', b = a'\}$$
$$E_1 \sqcup_{ufs} E_2 = \emptyset$$
$$E_1 \sqcup_{cufs} E_2 \supset \{f(C[a], C[b]) = f(C[a'], C[b']) \mid C \text{ is any context } \}$$

Here f is a binary symbol assumed commutative in the *cufs* theory. Note that $E_1 \sqcup_{cufs} E_2$ contains equalities like $f(a, b) = f(a', b')$ and is not finitely representable using ground equations even though $E_1 \sqcup_{ufs} E_2$ is finite.

Context-Sensitive Join Algorithm: Precise inter-procedural program analysis requires computing "context-sensitive procedure summaries", that is, invariants parameterized by the inputs so that given an instantiation for the inputs, the invariant can be instantiated to the most precise result for that input. Reps, Horwitz, and Sagiv described a general way to accomplish this for a simple class of data-flow analyses [16] . It is not clear how to do this in general for any abstract interpretation. The real challenge is in building an appropriate data-structure and a join algorithm for it that is context sensitive. For example, consider the following sets of equations E_1 and E_2.

$$E_1 = \{a = a', b = F(a')\} \qquad E_2 = \{a = b', b = F(b')\}$$
$$E_1 \sqcup E_2 = \{b = F(a)\}$$
$$E_1[a' = b'] \sqcup E_2[a' = b'] = \{a = a', b = F(a)\}$$
$$(E_1 \sqcup E_2)[a' = b'] = \{b = F(a)\}$$

This example illustrates that our join algorithm is not context-sensitive since it represents $E_1 \sqcup E_2$ as $\{b = F(a)\}$ which when instantiated in the context $a' = b'$ does not yield the most precise result. This suggests that a different data structure is required to obtain a context-sensitive join algorithm. Recently, Olm and Seidl have described a context-sensitive join algorithm for the theory of linear arithmetic with equality [13]. Their data structure is very different from the one used in Karr's join algorithm [10], which is not context-sensitive.

Combining Join Algorithms: Combining the join algorithm for UFS with the one for linear arithmetic (*la*) will give a join algorithm for the combined theory (*la_ufs*), which can be used to analyze programs with arrays and pointers. There are some nice results in the literature for combining decision procedures for different theories [14] , but none for combining join algorithms. Consider, for example, the following sets of equations E_1 and E_2.

$$E_1 = \{a = a', b = b'\} \qquad E_2 = \{a = b', b = a'\}$$
$$E_1 \sqcup_{la} E_2 = \{a + b = a' + b'\} \qquad E_1 \sqcup_{ufs} E_2 = \emptyset$$
$$E_1 \sqcup_{la\_ufs} \subset \{\forall i \geq 0, f^i a + f^i b = f^i a' + f^i b'\}$$

Here f is uninterpreted. Note that $E_1 \sqcup_{la\_ufs} E_2$ does not even admit finite presentation using ground equations. However, it may be possible to obtain a relatively complete join algorithm for the combined abstraction.

6 Conclusion

This paper explores the closure properties of congruence closure under the join operation. We show that the congruence closure representation is neither expressive enough nor compact enough to be able to represent the result of a join in the theory of UFS. The product of two congruence closures is related to the join and we show that it indeed provides a complete algorithm for certain special cases. This generalizes the known specific case of unification closures.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on POPL*, pages 1–11. ACM, 1988.
2. L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *J. of Automated Reasoning*, 31(2):129–168, 2003.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM PLDI '03*, pages 196–207, 2003.
4. P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, June 1997.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.
6. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth ACM Symposium on POPL*, pages 84–96, 1978.
7. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1990.
8. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *11th Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 212–227. Springer, 2004.
9. D. Kapur. Shostak's congruence closure as completion. In *Rewriting Techniques and Applications, RTA 1997*, pages 23–37. Springer-Verlag, 1997. LNCS 1103.
10. M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.
11. D. Kozen. Partial automata and finitely generated congruences: an extension of Nerode's theorem. In R. Shore, editor, *Proc. Conf. Logical Methods in Math. and Comp. Sci.*, 1992. Also Tech. Rep. PB-400, Comp. Sci. Dept., Aarhus Univ., 1992.
12. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.
13. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symposium on POPL*, pages 330–341. ACM, Jan. 2004.
14. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
15. G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, Apr. 1980.
16. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM Symposium on POPL*, pages 49–61. ACM, 1995.

17. E. Rodriguez-Carbonell and D. Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *11th Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*. Springer, 2004.
18. O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *SAS*, volume 1694 of *LNCS*, pages 232–247, 1999.
19. S. Vagvolgyi. Intersection of finitely generated congruences over term algebra. *Theoretical Computer Science*, 300:209–234, 2003.