Abstract Congruence Closure and Specializations*

Leo Bachmair and Ashish Tiwari

Department of Computer Science
State University of New York
Stony Brook, NY 11794-4400, U.S.A
{leo,astiwari}@cs.sunysb.edu

Abstract. We use the uniform framework of *abstract congruence closure* to study the congruence closure algorithms described by Nelson and Oppen [9], Downey, Sethi and Tarjan [7] and Shostak [11]. The descriptions thus obtained abstracts from certain implementation details while still allowing for comparison between these different algorithms. Experimental results are presented to illustrate the relative efficiency and explain differences in performance of these three algorithms. The transition rules for computation of abstract congruence closure are obtained from rules for *standard completion* enhanced with an *extension* rule that enlarges a given signature by new constants.

1 Introduction

Algorithms to compute "congruence closure" have typically been described in terms of directed acyclic graphs (dags) representing a set of terms, and a union-find data structure storing an equivalence relation on the vertices of this graph. In this paper, we abstractly describe some of these algorithms while still maintaining the "sharing" and "efficiency" offered by the data structures. This is achieved through the concept of an *abstract congruence closure*, c.f. [2, 3].

A key idea of abstract congruence closure is the use of new constants as names for subterms which yields a concise and simplified term representation. Consequently, complicated term orderings are no longer necessary or even applicable. There usually is a trade-off between the simplicity of terms thus obtained and the loss of term structure. In this paper, we get a middle ground where we keep the term structure as much as possible while still using extensions to obtain a simplified term representation. The paper also illustrates the use of an extended signature as a formalism to model and subsequently reason about data structures like the term dags, which are based on the idea of structure sharing.

In Section 2 we review the description of abstract congruence closure as a set of transition rules [2, 3]. The transition rules are derived from standard completion [1] enhanced with extension and suitably modified for the ground

* The research described in this paper was supported in part by the National Science Foundation under grant CCR-9902031.

case. Taking such an abstract view allows for a better understanding of the various graph-based congruence closure algorithms (Section 3), and also suggests new efficient procedures for constructing congruence closures (Section 4).

Preliminaries

Given a set $\Sigma = \cup_n \Sigma_n$ of function symbols and constants–called a *signature*–the set of (ground) terms $\mathcal{T}(\Sigma)$ over Σ is the smallest set containing Σ_0 and such that $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$ whenever $f \in \Sigma_n$ and $t_i \in \mathcal{T}(\Sigma)$. The index n of the set Σ_n to which a function symbol f belongs is called the *arity* of the symbol f. Elements of arity 0 are called *constants*. A symbol $f \in \Sigma_k$ of arity k is also said to be a k-ary function symbol. The symbols $s, t, u, \ldots$ are used to denote terms in $\mathcal{T}(\Sigma)$; $f, g, \ldots$, function symbols. We write $t[s]$ to indicate that a term t contains s as a subterm and (ambiguously) denote by $t[u]$ the result of replacing a particular occurrence of s by u. A subterm of a term t is called *proper* if it is distinct from t.

An *equation* is a pair of terms, written as $s \approx t$. The *replacement* or *single-step rewrite relation*[1] $\rightarrow_E$ induced by a set of ground (or variable-free) equations E is defined by: $u[l] \rightarrow_E u[r]$ if, and only if, $l \approx r$ is in E. If $\rightarrow$ is a binary relation, then $\leftarrow$ denotes its inverse, $\leftrightarrow$ its symmetric closure, $\rightarrow^+$ its transitive closure and $\rightarrow^*$ its reflexive-transitive closure. Thus, $\leftrightarrow^*_E$ denotes the *congruence relation*[2], which is the same as the *equational theory* when E is ground, induced by a set E of ground equations. Equations are often called *rewrite rules*, and a set E a *rewrite system*, if one is interested particularly in the *rewrite relation* $\rightarrow^*_E$ rather than the equational theory $\leftrightarrow^*_E$.

If E is a set of equations, we write $E[s]$ to denote that the term s occurs as a subterm of some equation in E, and (ambiguously) use $E[t]$ to denote the set of equations obtained by replacing an occurrence of s in E by t.

A term t is in *normal form* with respect to a rewrite system R if there is no term t' such that $t \rightarrow_R t'$. We write $s \rightarrow^!_R t$ to indicate that t is a R-normal form of s. A rewrite system R is said to be (ground) *confluent* if every (ground) terms t has at most one normal form, i.e., if there exist s, s' such that $s \leftarrow^*_R t \rightarrow^*_R s'$, then, $s \rightarrow^*_R \circ \leftarrow^*_R s'$. It is *terminating* if there exists no infinite sequence $s_0 \rightarrow_R s_1 \rightarrow_R s_2 \cdots$ of terms. Rewrite systems that are (ground) confluent and terminating are called (ground) *convergent*.

2 Abstract Congruence Closure

We first review the concept of an abstract congruence closure [2, 3]. Let Σ be a signature and K be a set of constants disjoint from Σ. A *D-rule* (with respect to Σ and K) is a rewrite rule of the form $t \rightarrow c$ where t is a term from the set

[1] There is no difference between the replacement relation and the rewrite relation in the ground case.

[2] A congruence relation is a reflexive, symmetric and transitive relation on terms that is also a replacement relation.

$\mathcal{T}(\Sigma \cup K) - K$ and c is a constant in K[3]. A *C-rule* (with respect to K) is a rule $c \rightarrow d$, where c and d are constants in K. For example, if $\Sigma_0 = \{a, b, f\}$, and $E_0 = \{a \approx b, ffa \approx fb\}$ then $D_0 = \{a \rightarrow c_0,\ b \rightarrow c_1,\ ffa \rightarrow c_2,\ fb \rightarrow c_3\}$ is a set of D-rules over Σ_0 and $K_0 = \{c_0, c_1, c_2, c_3\}$. Original equations in E_0 can now be simplified using D_0 to give $C_0 = \{c_0 \approx c_1, c_2 \approx c_3\}$. The set $D_0 \cup C_0$ may be viewed as an alternative representation of E_0 over an extended signature. The equational theory presented by $D_0 \cup C_0$ is a conservative extension of the theory E_0. This reformulation of the equations E_0 in terms of an extended signature is (implicitly) present in all congruence closure algorithms, see Section 3.

A constant c in K is said to *represent* a term t in $\mathcal{T}(\Sigma \cup K)$ (via the rewrite system R) if $t \leftrightarrow_R^* c$. A term t is *represented* by R if it is represented by some constant in K via R. For example, the constant c_2 represents the term ffa via D_0.

Definition 1. *Let Σ be a signature and K be a set of constants disjoint from Σ. A ground rewrite system $R = D \cup C$ of D-rules and C-rules over the signature $\Sigma \cup K$ is said to be an* (abstract) congruence closure *(with respect to Σ and K) if (i) each constant $c \in K$ that is in normal form with respect to R, represents some term $t \in \mathcal{T}(\Sigma)$ via R, and (ii) R is ground convergent.*

If E is a set of ground equations over $\mathcal{T}(\Sigma \cup K)$ and in addition R is such that (iii) for all terms s and t in $\mathcal{T}(\Sigma)$, $s \leftrightarrow_E^* t$ if, and only if, $s \rightarrow_R^* \circ \leftarrow_R^* t$, then R will be called an (abstract) congruence closure for E.

Condition (i) essentially states that no superfluous constants are introduced; condition (ii) ensures that equivalent terms have the same representative; and condition (iii) implies that R is a conservative extension of the equational theory induced by E over $\mathcal{T}(\Sigma)$.

The rewrite system $R_0 = D_0 \cup \{c_0 \rightarrow c_1,\ c_2 \rightarrow c_3\}$ above is not a congruence closure for E_0, as it is not ground convergent. But we can transform R_0 into a suitable rewrite system, using a completion-like process described in more detail below, to obtain a congruence closure

$$R_1 = \{a \rightarrow c_1,\ b \rightarrow c_1,\ fc_1 \rightarrow c_3,\ fc_3 \rightarrow c_3,\ c_0 \rightarrow c_1,\ c_2 \rightarrow c_3\}.$$

Construction of Congruence Closures

We next present a general method for construction of congruence closures. Our description is fairly abstract, in terms of transition rules that manipulate triples (K, E, R), where K is the set of constants that extend the original fixed signature Σ, E is the set of ground equations (over $\Sigma \cup K$) yet to be processed, and R is the set of C-rules and D-rules that have been derived so far. Triples represent *states* in the process of constructing a congruence closure. Construction starts from *initial state* $(\emptyset, E, \emptyset)$, where E is a given set of ground equations.

[3] The definition of a D-rule is more general than the definition presented in [2, 3] as it allows for arbitrary non-constant terms on the left-hand side.

The transition rules can be derived from those for standard completion as described in [1], with some differences. In particular, (i) application of the transition rules is guaranteed to terminate, and (ii) a convergent system is constructed over an extended signature. The transition rules do *not* require any reduction ordering[4] on terms in $\mathcal{T}(\Sigma)$, but only only a simple ordering $\succ$ on terms in $\mathcal{T}(\Sigma \cup U)$[5] where U is an infinite set of constants from which new constants $K \subset U$ are chosen. In particular, if we assume $\succ_U$ is any ordering on the set U, then $\succ$ is defined as: $c \succ d$ if $c \succ_U d$ and $t \succ c$ if $t \to c$ is a D-rule. In this paper, the set $U = \{c_0, c_1, c_2, \ldots\}$, and we will assume $c_i \succ_U c_j$ iff $i < j$.

A key transition rule introduces new constants as names for subterms.

$$\textbf{Ext}\text{ension:} \qquad \frac{(K, E[t], R)}{(K \cup \{c\}, E[c], R \cup \{t \to c\})}$$

where $t \to c$ is a D-rule, t is a term occurring in (some equation in) E, and $c \notin \Sigma \cup K$.

Following three rules are identical to the corresponding rules for standard completion.

$$\textbf{Sim}\text{plification:} \quad \frac{(K, E[t], R \cup \{t \to c\})}{(K, E[c], R \cup \{t \to c\})}$$

where t occurs in some equation in E.

It is fairly easy to see that by repeated application of extension and simplification, any equation in E can be reduced to an equation that can be oriented by the ordering $\succ$.

$$\textbf{Ori}\text{entation:} \quad \frac{(K \cup \{c\}, E \cup \{t \approx c\}, R)}{(K \cup \{c\}, E, R \cup \{t \to c\})}$$

if $t \succ c$.

Trivial equations may be deleted.

$$\textbf{Del}\text{etion:} \quad \frac{(K, E \cup \{t \approx t\}, R)}{(K, E, R)}$$

In the case of completion of ground equations, deduction steps can all be replaced by suitable simplification steps. In particular, most of the deduction steps can be described by collapse, and hence, the deduction rule considers only simple forms of overlap.

$$\textbf{Ded}\text{uction:} \quad \frac{(K, E, R \cup \{t \to c,\ t \to d\})}{(K, E \cup \{c \approx d\}, R \cup \{t \to d\})}$$

[4] An *ordering* is any irreflexive and transitive relation on terms. A *reduction ordering* is an ordering that is also a well-founded replacement relation.

[5] Terms in $\mathcal{T}(\Sigma)$ are uncomparable by $\succ$.

In our case the usual side condition in the collapse rule, which refers to the *encompassment ordering*, can easily be stated in terms of the subterm relation.

$$\textbf{Col}\text{lapse:} \qquad \frac{(K, E, R \cup \{s[t] \to d,\ t \to c\})}{(K, E, R \cup \{s[c] \to d,\ t \to c\})}$$

if t is a proper subterm of s.

As in standard completion the simplification of right-hand sides of rules in R by other rules is optional and not necessary for correctness. The right-hand side term in any rule in R is always a constant.

$$\textbf{Com}\text{position:} \qquad \frac{(K, E, R \cup \{t \to c,\ c \to d\})}{(K, E, R \cup \{t \to d,\ c \to d\})}$$

We use the symbol $\vdash$ to denote the one-step transition relation on states induced by the above transition rules. A *derivation* is a sequence of states $(K_0, E_0, R_0) \vdash (K_1, E_1, R_1) \vdash \cdots$.

Example 1. Consider the set of equations $E_0 = \{a \approx b,\ fffa \approx fb\}$. An abstract congruence closure for E_0 can be derived from $(K_0, E_0, R_0) = (\emptyset, E_0, \emptyset)$ as follows:

i	Constants K_i	Equations E_i	Rules R_i	Transition Rule
0	$\emptyset$	E_0	$\emptyset$	
1	$\{c_0\}$	$\{c_0 \approx b, fffa \approx fb\}$	$\{a \to c_0\}$	**Ext**
2	$\{c_0\}$	$\{fffa \approx fb\}$	$\{a \to c_0, b \to c_0\}$	**Ori**
3	$\{c_0\}$	$\{ffc_0 \approx fc_0\}$	$\{a \to c_0, b \to c_0\}$	$\textbf{Sim}^2$
4	$\{c_0, c_1\}$	$\{fc_1 \approx fc_0\}$	$R_3 \cup \{fc_0 \to c_1\}$	**Ext**
5	$\{c_0, c_1\}$	$\{fc_1 \approx c_1\}$	$R_3 \cup \{fc_0 \to c_1\}$	**Sim**
6	K_5	$\{\}$	$R_5 \cup \{fc_1 \to c_1\}$	**Ori**

The rewrite system R_6 is the required congruence closure.

The correctness of the transition rules presented here can be established in a way similar to the correctness of the transition rules for computing a congruence closure modulo associativity and commutativity [3]. The differences arise from the more general definition of D-rules, and the lack of any associative and commutative functions here.

The set of transition rules presented above are sound in the following sense: if $(K_0, E_0, R_0) \vdash (K_1, E_1, R_1)$, then, for all terms s and t in $\mathcal{T}(\Sigma \cup K_0)$, $s \leftrightarrow^*_{E_1 \cup R_1} t$ if and only if $s \leftrightarrow^*_{E_0 \cup R_0} t$. Additionally, let K_0 be a *finite* set of constants (disjoint from Σ), E_0 be a *finite* set of equations (over $\Sigma \cup K_0$), and R_0 be a *finite* set of D-rules and C-rules such that for every C-rule $c \to d \in R_0$, we have $c \succ_U d$. Then, any derivation starting from (K_0, E_0, R_0) is finite. If $(K_0, E_0, R_0) \vdash^* (K_m, E_m, R_m)$, then R_m is terminating. We call a state (K, E, R) *final* if no transition rule (except possibly composition) is applicable.

Theorem 1. *Let Σ be a signature and K_1 a finite set of constants disjoint from Σ. Let E_1 be a finite set of equations over $\Sigma \cup K_1$ and R_1 a finite set of D-rules and C-rules such that for every $c \in K_1$ represents some term $t \in \mathcal{T}(\Sigma)$ via $E_1 \cup R_1$, and $c \succ_U d$ for every C-rule $c \to d$ in R_1. If (K_n, E_n, R_n) is a final state such that $(K_1, E_1, R_1) \vdash^* (K_n, E_n, R_n)$, then $E_n = \emptyset$ and R_n is an abstract congruence closure for $E_1 \cup R_1$ (over Σ and K_1).*

3 Congruence Closure Strategies

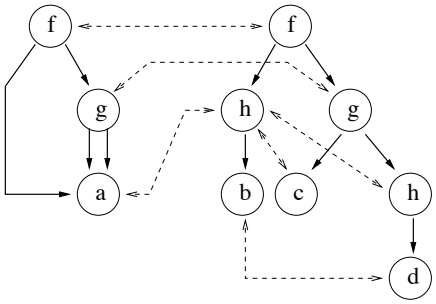
The literature abounds with various implementations of congruence closure algorithms. We next describe the algorithms in [7], [9] and [11] as specific variants of our general abstract description. That is, we provide a description of these algorithms (modulo some implementation details) using abstract congruence closure transition rules.

Term directed acyclic graphs (dags) is a common data structure used to implement algorithms that work with terms over some signature—such as the congruence closure algorithm. In fact, many algorithms that have been described for congruence closure assume that the input is an equivalence relation on vertices of a given dag, and the desired output is an equivalence on the same dag that is defined by the congruence relation.

Figure 1 illustrate how a given term dag is (abstractly) represented using D-rules. The solid lines represent *subterm* edges, and the dashed lines represent a binary relation on the vertices. We have a D-rule corresponding to each vertex, and a C-rule for each dashed edge. Note that the D-rules corresponding to a conventional term dag representation are all of a special form $f(c_1, \ldots, c_k) \to c$, where $f \in \Sigma$ is a k-ary function symbol, and $c_1, \ldots, c_k, c$ are all new constants. Such rules will be called *simple* D-rules. The definition of D-rules given in Section 2 is more general, and allows for arbitrary terms on the left-hand sides. In a sense this corresponds to storing *contexts*, rather than just symbols from Σ, in each node (of the term dag). This is an attempt to keep as much of the term structure information as possible and still get advantages offered by a simplified term representation via extensions.

We need to specify a U set and an ordering $\succ_U$ on this set. Since elements of U serve only as names, we can choose U to be any countable set of symbols. An ordering $\succ_U$ need not be specified a-priori but can be defined on-the-fly as the derivation proceed. (The ordering has to be extended so that the irreflexivity and transitivity properties are preserved).

Traditional congruence closure algorithms also employ other data structures such as the following:
(i) Input dag: Starting from the state $(\emptyset, E_0, \emptyset)$, if we apply extension and simplification using strategy $(\mathbf{Ext} \circ \mathbf{Sim}^*)^*$ and making sure we create only simple D-rules, we finally get to a state (K_1, E_1, D_1) where all equations in E_1 are of the form $c \approx d$, for $c, d \in K_1$. The set D_1, then, represents the input dag and E_1 represents the (input) equivalence on the vertices of this dag. Note that due to eager simplification, we obtain representation of a dag with maximum possible



D-rules representing the term dag:

$$\begin{array}{rrr} a \to c_1 & gc_1c_1 \to c_2 & fc_1c_2 \to c_3 \\ b \to c_4 & hc_4 \to c_5 & c \to c_6 \\ d \to c_7 & hc_7 \to c_8 & gc_6c_8 \to c_9 \\ fc_5c_9 \to c_{10} & & \end{array}$$

C-rules representing the relation on vertices:

$$\begin{array}{lll} c_1 \approx c_5 & c_2 \approx c_9 & c_3 \approx c_{10} \\ c_4 \approx c_7 & c_6 \approx c_5 & c_5 \approx c_8 \end{array}$$

Fig. 1. A term dag and a relation on its vertices

sharing. For example, if $E_0 = \{a \approx b, fffa \approx fb\}$, then $K_1 = \{c_0, c_1, c_2, c_3, c_4\}$, $E_1 = \{c_0 \approx c_1, c_3 \approx c_4\}$ and $R_1 = \{a \to c_0, b \to c_1, fc_0 \to c_2, fc_2 \to c_3, fc_1 \to c_4\}$.
(ii) Signature table: The *signature* table (indexed by vertices of the input dag) stores a *signature*[6] for some or all vertices. Clearly, the signatures are fully left-reduced D-rules.
(iii) Use table: The *use* table (also called predecessor list) is a mapping from the constant c to the set of all vertices whose signature contains c. This translates, in our presentation, to a method of indexing the set of D-rules.
(iv) Union Find: The union-find data structure that maintains equivalence classes on the set of vertices is represented by the set of C rules. If we apply orientation and simplification to the state (K_1, E_1, D_1) described above, using the strategy $(\mathbf{Ori} \circ \mathbf{Sim}^*)^*$, we obtain a state $(K_1, \emptyset, D_1 \cup C_1)$. The set C_1 is a representation of the Union-Find structure capturing the input equivalence on vertices. Continuing with the same example, C_1 would be the set $\{c_0 \to c_1, c_3 \to c_4\}$.

We note that, D-rules serve a two-fold purpose: they represent the input term dag, and also a signature table. We shall also note that Composition is used only implicitly in the various algorithms via path-compression on the union-find structure.

Shostak's Method

Shostak's congruence closure procedure was first described using *simple* D-rules and C-rules by Kapur [8]. We show here that Shostak's congruence closure procedure is a specific strategy over the general transition rules for abstract congruence closure presented here.

Shostak's congruence closure is *dynamic*: it can accept new equations after it has processed some equations, and can incrementally take care of the new

[6] The signature of a term $f(t_1, \ldots, t_k)$ is defined as $f(c_1, \ldots, c_k)$ where c_i is the name of the equivalence class containing term t_i.

equation. Its input state is $(\emptyset, E_0, \emptyset)$. Shostaks procedure can be described (at a fairly abstract level) as:

$$\mathbf{Shos} = ((\mathbf{Sim}^* \circ \mathbf{Ext}^*)^* \circ (\mathbf{Del} \cup \mathbf{Ori}) \circ (\mathbf{Col} \circ \mathbf{Ded}^*)^*)^*$$

which is implemented as (i) pick an equation $s \approx t$ from the E-component, (ii) use simplification to normalize the term s to a term s' (iii) use extension to create *simple* D-rules for subterms of s' until s' reduces to a constant, say c, whence extension is no longer applicable. Perform steps (ii) and (iii) on the other term t as well to get a constant d. (iv) if c and d are identical then apply deletion (and continue with (i)), and if not, create a C-rule using orientation. (v) Once we have a new C-rule, perform all possible collapse step by this new rule, where each collapse step is followed by all the resulting deduction steps arising out of that collapse. The whole process is now repeated starting from step (i).

Shostak's procedure uses indexing based on the idea of the *use*() list. This *use*() based indexing is used to identify all possible collapse applications.

If the E-component of the state is empty while attempting to apply step (i), Shostak's procedure halts. It is fairly easy to observe that Shostak's procedure halts in a *final* state. Hence, Theorem 1 establishes that the R-component of Shostak's halting state contains a convergent system and is an abstract congruence closure.

Example 2. We use the set E_0 used in Example 1 of Section 2 to illustrate Shostak's method. We show some of the important intermediate steps of a Shostak derivation.

i	Constants K_i	Equations E_i	Rules R_i	Transition
0	$\emptyset$	E_0	$\emptyset$	
1	$\{c_0, c_1\}$	$\{fffa \approx fb\}$	$\{a \to c_0,\ b \to c_1,\ c_0 \to c_1\}$	$\mathbf{Ext}^2 \circ \mathbf{Ori}$
2	$\{c_0, c_1\}$	$\{fffc_1 \approx fb\}$	$\{a \to c_0,\ b \to c_1,\ c_0 \to c_1\}$	$\mathbf{Sim}$
3	$\{c_0, \ldots, c_3\}$	$\{c_3 \approx fb\}$	$R_2 \cup \{fc_1 \to c_2,\ fc_2 \to c_3\}$	$\mathbf{Ext}^2$
4	$\{c_0, \ldots, c_3\}$	$\{c_3 \approx c_2\}$	R_3	$\mathbf{Sim}^2$
5	$\{c_0, \ldots, c_3\}$	$\emptyset$	$R_4 \cup \{c_3 \to c_2\}$	$\mathbf{Ori}$

The Downey-Sethi-Tarjan Algorithm

The Downey, Sethi and Tarjan [7] procedures assumes that the input is a dag and an equivalence relation on its vertices, which, in our language, means that the starting state for this procedures is $(K_1, \emptyset, D_1 \cup C_1)$, where D_1 represents the input dag and C_1 represents the initial equivalence. It can be succinctly abstracted as:

$$\mathbf{DST} = ((\mathbf{Col} \circ (\mathbf{Ded} \cup \{\epsilon\}))^* \circ (\mathbf{Sim}^* \circ (\mathbf{Del} \cup \mathbf{Ori}))^*)^*$$

where ϵ is the null transition rule. This strategy is implemented as follows (i) if any collapse rule is applicable, it is applied and if, as a result any new deduction step is possible, it is done. This is repeated until no more collapse steps are

possible. (ii) if no collapse steps are possible, then each C-equation in the E-component is picked up sequentially, fully-simplified (simplification) and then either deleted (deletion) or oriented (orientation).

Although the above description captures the essence of the Downey, Sethi and Tarjan procedure, a few implementation details need to be pointed out. Firstly, the Downey, Sethi and Tarjan procedure keeps the original dag (represented by D_1) intact[7], but changes signatures in a signature table. Hence, in the actual implementation described in [7], the $(\mathbf{Col} \circ (\mathbf{Ded} \cup \{\epsilon\}))^*$ strategy is applied by: (i) deleting all signatures that will be changed, i.e., deleting all D-rules which can be collapsed; (ii) computing new signatures using the *original* copy of the signatures stored in the form of the dag D_1; and, finally, (iii) inserting the newly computed signatures into the signature table and checking for possible deduction steps. Our description achieves the same end result, but, by doing fewer inferences.

Secondly, in the Downey, Sethi and Tarjan procedure, for efficiency, an equation $c \approx d$ is oriented to $c \to d$ if the c occurs fewer times than d in the signature table. This is done to minimize the number of collapse steps. Additionally, indexing based on the *use*() tables is used for efficiently implementing the specific strategy.

Let $(K_1, \emptyset, D_1 \cup C_1) \vdash^! (K_n, E_n, D_n \cup C_n)$ be a derivation using the DST strategy. Then, it is easily seen that the state $(K_n, E_n, D_n \cup C_n)$ is a final state, and hence the set $D_n \cup C_n$ is convergent, and also an abstract congruence closure. We remark here that D_n holds the information that is contained in the signature table, and C_n holds information in the union-find structure. The set C_n is usually considered the output of the Downey, Sethi and Tarjan procedure.

Example 3. We illustrate the Downey-Sethi-Tarjan algorithm by using the same set of equations E_0, used in Example 1 of Section 2. The start state is $(K_1, \emptyset, D_1 \cup C_1)$ where $K = \{c_0, \ldots, c_4\}$, $D_1 = \{a \to c_0,\ b \to c_1,\ fc_0 \to c_2,\ fc_2 \to c_3,\ fc_1 \to c_4\}$, and, $C_1 = \{c_0 \to c_1,\ c_3 \to c_4\}$.

i	Consts K_i	Eqns E_i	Rules R_i	Transition
1	K_1	$\emptyset$	$D_1 \cup C_1$	
2	K_1	$\emptyset$	$\{a \to c_0,\ b \to c_1,\ fc_1 \to c_2,$ $fc_2 \to c_3,\ fc_1 \to c_4\} \cup C_1$	**Col**
3	K_1	$\{c_2 \approx c_4\}$	R_2	**Ded**
4	K_1	$\emptyset$	$R_3 - \{fc_1 \to c_2\} \cup \{c_4 \to c_2\}$	**Ori**

Note that $c_4 \approx c_2$ was oriented in a way that no further collapses were needed thereafter.

The Nelson-Oppen Procedure

The Nelson-Oppen procedure is not exactly a completion procedure and it does *not* generate a congruence closure in our sense. The initial state of the Nelson-

[7] We could make a copy of the original D_1 rules and not change them, while keeping a separate copy as the signatures.

Oppen procedure is given by the tuple (K_1, E_1, D_1), where D_1 is the input dag, and E_1 represents an equivalence on vertices of this dag. The sets K_1 and D_1 remain unchanged in the Nelson-Oppen procedure. In particular, the inference rule used for deduction is different from the conventional deduction rule[8].

$$\textbf{NODed}\text{uction: } \frac{(K, E, D \cup C)}{(K, E \cup \{c \approx d\}, D \cup C)}$$

if there exist two D-rules $f(c_1, \ldots, c_k) \to c$, and, $f(d_1, \ldots, d_k) \to d$ in the set D; and, $c_i \to^!_C \circ \leftarrow^!_C d_i$, for $i = 1, \ldots, k$.

The Nelson-Oppen procedure can now be (at a certain abstract level) represented as:

$$\mathbf{NO} = (\mathbf{Sim}^* \circ (\mathbf{Ori} \cup \mathbf{Del}) \circ \mathbf{NODed}^*)^*$$

which is applied in the following sense: (i) select a C-equation $c \approx d$ from the E-component, (ii) simplify the terms c and d using simplification steps until the terms can't be simplified any more, (iii) either delete, or orient the simplified C-equation, (iv) apply the NODeduction rule until there are no more non-redundant applications of this rule, (v) if the E-component is empty, then we stop, otherwise continue with step (i).

Certain details like the fact that newly added equations to the set E are chosen before the old ones in an application of orientation and indexing based on the *use*() table, are abstracted away in this description.

Using the Nelson-Oppen strategy, assume we get a derivation $(K_1, E_1, D_1) \vdash^*_{\mathbf{NO}} (K_n, E_n, D_n \cup C_n)$. One consequence of using a non-standard deduction rule, NODeduction, is that the resulting set $D_n \cup C_n = D_1 \cup C_n$ need not necessarily be convergent, although the the rewrite relation D_n/C_n [6] is convergent.

Example 4. Using the same set E_0 as equations, we illustrate the Nelson-Oppen procedure. The initial state is given by (K_1, E_1, D_1) where $K_1 = \{c_0, c_1, c_2, c_3, c_4\}$; $E_1 = \{c_0 \approx c_1,\ c_3 \approx c_4\}$; and, $D_1 = \{a \to c_0,\ b \to c_1,\ fc_0 \to c_2,\ fc_2 \to c_3,\ fc_1 \to c_4\}$.

i	Constants K_i	Equations E_i	Rules R_i	Transition
1	K_1	E_1	D_1	
2	K_1	$\{c_3 \approx c_4\}$	$D_1 \cup \{c_0 \to c_1\}$	**Ori**
3	K_1	$\{c_2 \approx c_4,\ c_3 \approx c_4\}$	R_2	**NODed**
4	K_1	$\{c_3 \approx c_4\}$	$R_2 \cup \{c_2 \to c_4\}$	**Ori**
5	K_1	$\emptyset$	$R_4 \cup \{c_3 \to c_4\}$	**Ori**

Consider deciding the equality $fa \approx ffb$. Even though $fa \leftrightarrow^*_{E_0} ffb$, the terms fa and ffb have distinct normal forms with respect to R_5. But terms in the original term universe have identical normal forms.

[8] This rule performs deduction modulo C-equations, i.e., we compute critical pairs between D-rules modulo the congruence induced by C-equations. Hence, the Nelson-Oppen procedure can be described as an *extended completion* [6] (or completion modulo C-equations) method over an extended signature.

4 Experimental Results

We have implemented five congruence closure algorithms, including those proposed by Nelson and Oppen (NO) [9], Downey, Sethi and Tarjan (DST) [7], and Shostak [11], and two algorithms based on completion—one with an indexing mechanism (IND) and the other without (COM). The implementations of the first three procedures are based on the representation of terms by directed acyclic graphs and the representation of equivalence classes by a union-find data structure. The completion procedure COM uses the following strategy:

$$((\mathbf{Sim}^* \circ \mathbf{Ext}^*)^* \circ (\mathbf{Del} \cup \mathbf{Ori}) \circ (\mathbf{Com} \circ \mathbf{Col})^* \circ \mathbf{Ded}^*)^*.$$

The indexed variant IND uses a slightly different strategy

$$((\mathbf{Sim}^* \circ \mathbf{Ext}^*)^* \circ (\mathbf{Del} \cup \mathbf{Ori}) \circ (\mathbf{Col} \circ \mathbf{Com} \circ \mathbf{Ded})^*)^*.$$

Indexing in the case of completion refers to the use of suitable data structures to efficiently identify which D-rules contain specified constants.

In a first set of experiments, we assume that the input is a set of equations presented as pairs of trees (representing terms). We added a preprocessing step to the NO and DST algorithms to convert the given input terms into a dag and initialize the other required data-structures. The other three algorithms interleave construction of a dag with deduction steps. The published descriptions DST and NO do not address the construction of a dag. Our implementation maintains the list of terms that have been represented in the dag in a hash table and creates a new node for each term not yet represented. We present below a sample of our results to illustrate some of the differences between the various algorithms.

The input set of equations E can be classified based on: (i) the size of the input and the number of equations, (ii) the number of equivalence classes on terms and subterms of E, and, (iii) the size of the *use* lists. The first set of examples are relatively simple and developed by hand to highlight strengths and weaknesses of the various algorithms. Example (a)[9] contains five equations that induce a single equivalence class. Example (b) is the same as (a), except that it contains five copies of all the equations. Example (c)[10] requires slightly larger *use* lists. Finally, example (d)[11] consists of equations that are oriented in the "wrong" way.

In Table 1 we compare the different algorithms by their total running time, *including* the preprocessing time. The times shown are the averages of several runs on a Sun Ultra workstation under similar load conditions. The time was computed using the *gettimeofday* system call.

[9] The equation set is $\{f^2(a) \approx a, f^{10}(a) \approx f^{15}(b), b \approx f^5(b), a \approx f^3(a), f^5(b) \approx b\}$.

[10] The equation set is $\{g(a,a,b) \approx f(a,b)$, $gabb \approx fba$, $gaab \approx gbaa$, $gbab \approx gabb$, $gbba \approx gbab$, $gaaa \approx faa$, $a \approx c$, $c \approx d$, $d \approx e$, $b \approx c1$, $c1 \approx d1$, $d1 \approx e1\}$.

[11] The set is $\{g(f^i(a), h^{10}(b)) \approx g(a,b), i = \{1, \cdots, 25\}, h^{47}(b) \approx b, b \approx h^{29}(b), h(b) \approx c0, c0 \approx c1, c1 \approx c2, c2 \approx c3, c3 \approx c4, c4 \approx a, a \approx f(a)\}$.

	Eqns	Vert	Class	DST	NO	SHO	COM	IND
Ex.a	5	27	1	1.286	1.640	0.281	0.606	0.409
Ex.b	20	27	1	2.912	2.772	0.794	1.858	0.901
Ex.c	12	20	6	1.255	0.733	0.515	0.325	0.323
Ex.d	34	105	2	10.556	22.488	7.275	12.077	4.416

Table 1. Total running time (in milliseconds) for Examples $(a)--(d)$. *Eqns* refers to the number of equations; *Vert* to the number of vertices in the initial dag; and *Class* to the number of equivalence classes induced on the dag.

Table 2 contains similar comparisons for considerably larger examples consisting of randomly generated equations over a specified signature. Again we show total running time, including preprocessing time[12].

	Eqns	Vert	Σ_0	Σ_1	Σ_2	d	Class	DST	NO	SHO	IND
Ex.1	10000	17604	2	0	2	3	7472	11.087	3.187	10.206	13.037
Ex.2	5000	4163	2	1	1	3	3	2.276	306.194	3.092	0.774
Ex.3	5000	7869	3	0	1	3	2745	2.439	1.357	3.521	3.989
Ex.4	6000	8885	3	0	1	3	9	3.551	1152.652	52.353	7.069
Ex.5	7000	9818	3	0	1	3	1	4.633	1682.815	47.755	5.471
Ex.6	5000	645	4	2	0	23	77	1.224	1.580	0.371	0.363
Ex.7	5000	1438	10	2	0	23	290	1.452	3.670	0.392	0.374

Table 2. Total running time (in seconds) for randomly generated equations. The columns Σ_i denote the number of function symbols of arity i in the signature and d denotes the maximum term depth.

In Table 3 we show the time for computing a congruence closure assuming terms are already represented by a dag. In other words, we do not include the time it takes to create a dag. Note that we include no comparison with Shostak's method, as the dynamic construction of a dag from given term equations is inherent in this procedure. However, a comparison with a suitable strategy (in which all extension steps are applied before any deduction steps) of IND is possible. We denote by IND* indexed completion based on a strategy that first constructs a dag. The examples are the same as in Table 2.

Several observations can be drawn from these results. First, the Nelson-Oppen procedure NO is competitive only when few deduction steps are performed and thus the number of equivalence classes is large. This is because it uses a non-standard deduction rule, which forces the procedure to unnecessarily repeat the same deductions many times over in a single execution. Not surprisingly, straight-forward completion without indexing is also inefficient when

[12] Times for COM are not included as indexing is indispensable for larger examples.

	DST	NO	IND*
Ex.1	0.919	0.296	0.076
Ex.2	0.309	319.112	1.971
Ex.3	0.241	0.166	0.030
Ex.4	0.776	1117.239	7.301

	DST	NO	IND*
Ex.5	0.958	1614.961	9.770
Ex.6	0.026	0.781	0.060
Ex.7	0.048	2.470	0.176

Table 3. Running time (in seconds) when input is in a dag form.

many deduction steps are necessary. Indexing is of course a standard technique employed in all practical implementations of completion.

The running time of the DST procedure critically depends on the size of the hash table that contains the signatures of all vertices. If the hash table size is large, enough potential deductions can be detected in (almost) constant time. If the hash table size is reduced, to say 100, then the running time increased by a factor of up to 50. A hash table with 1000 entries was sufficient for our examples (which contained fewer than 10000 vertices). Larger tables did not improve the running times.

Indexed Completion, DST and Shostak's method are roughly comparable in performance, though Shostak's algorithm has some drawbacks. For instance, equations are always oriented from left to right. In contrast, Indexed Completion always orients equations in a way so as to minimize the number of applications of the collapse rule, an idea that is implicit in Downey, Sethi and Tarjan's algorithm. Example (*b*) illustrates this fact. More crucially, the manipulation of the *use* lists in Shostak's method is done in a convoluted manner due to which redundant inferences may be done when searching for the correct non-redundant ones[13]. As a consequence, Shostak's algorithm performs poorly on instances where *use* lists are large and deduction steps are many such as in Examples (*c*), 4 and 5.

Finally, we note that the indexing used in our implementation of completion is simple—with every constant c we associate a list of D-rules that contain c as a subterm. On the other hand DST maintains at least two different ways of indexing the signatures, which makes it more efficient when the examples are large and deduction steps are plenty. On small examples, the overhead to maintain the data structures dominates. This also suggests that the use of more sophisticated indexing schemes for indexed completion might improve its performance.

5 Related Work and Conclusion

Kapur [8] considered the problem of casting Shostak's congruence closure [11] algorithm in the framework of ground completion on rewrite rules. Our work has been motivated by the goal of formalizing not just one, but several congruence closure algorithms, so as to be able to better compare and analyze them.

[13] The description in Section 3 accurately reflects the logical aspects of Shostak's algorithm, but does not provide details on data structures like the *use* lists.

We suggest that, abstractly, congruence closure can be *defined* as a ground convergent system; and that this definition does not restrict the applicability of congruence closure. The rule-based abstract description of the logical aspects of the various published congruence closure algorithms leads to a better understanding of these methods. It explains the observed behaviour of implementations and also allows one to identify weaknesses in specific algorithms. Additionally, using the abstract rules, we can also get efficient implementation of completion based congruence closure procedure—one can effectively utilize the theory of redundancy to figure out and eliminate inferences which are not necessary, and moreover also use knowledge about efficient indexing mechanisms.

The concept of an abstract congruence closure is also relevant for describing applications that use congruence closure algorithms. Some of these applications include efficient normalization by rewrite systems [4, 2], computing a complete set of rigid E-unifiers [13], and combination of decision procedures [11]. The notion of an abstract congruence closure is naturally extended to handle presence of associative-commutative operators, and this application is described in [3]. We believe that theories other than associativity and commutativity can also be incorporated with the inference rules for abstract congruence closure.

Congruence closure has also been used to construct a convergent set of ground rewrite rules in polynomial time by Snyder [12] and other works. Plaisted et. al. [10] gave a *direct* method, not based on using congruence closure, for completing a ground rewrite system in polynomial time. Hence our work completes the missing link, by showing that congruence closure is nothing but ground completion. In fact, the process of transforming a set of rewrite rules over an extended signature (representing an abstract congruence closure) into a convergent set of rewrite rules over the original signature can be easily described by additional transition rules [3]. Our approach is different from that of Snyder, and can be used to obtain a *more efficient* implementation partly because Snyder's algorithm needs *two* passes of the congruence closure algorithm, whereas we would need to compute the abstract congruence closure just once.

The concept of an abstract congruence closure as detailed here and the rules for computation open up new frontiers too. For example, the transition rules presented in Section 2 can be naturally implemented in MAUDE [5]. Moreover, specific strategies, such as the ones presented in Section 3 can be encoded easily too. This might provide a basis for automatically verifying the correctness of congruence closure algorithms[14].

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments.

References

[1] L. Bachmair and N. Dershowitz. Equational inference, canonical proofs, and proof orderings. *J. ACM*, 41:236–276, 1994.

[14] Personal communication with Manuel Clavel.

[2] L. Bachmair, C. Ramakrishnan, I. Ramakrishnan, and A. Tiwari. Normalization via rewrite closures. In P. Narendran and M. Rusinowitch, editors, *10th Int. Conf. on Rewriting Techniques and Applications*, pages 190–204, 1999. LNCS 1631.

[3] L. Bachmair, I. Ramakrishnan, A. Tiwari, and L. Vigneron. Congruence closure modulo associativity and commutativity. In H. Kirchner and C. Ringeissen, editors, *Frontiers of Combining Systems, 3rd Intl Workshop FroCoS 2000*, pages 245–259, 2000. LNAI 1794.

[4] L. P. Chew. *Normal forms in term rewriting systems.* PhD thesis, Purdue University, 1981.

[5] M. Clavel and et. al. *Maude: Specification and Programming in Rewriting Logic.* http://maude.csl.sri.com/manual/, SRI International, Menlo Park, CA, 1999.

[6] N. Dershowitz and J. P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B: Formal Models and Semantics)*, Amsterdam, 1990. North-Holland.

[7] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *J. ACM*, 27(4):758–771, 1980.

[8] D. Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, pages 23–37, 1997. Vol. 1232 of *Lecture Notes in Computer Science*, Springer, Berlin.

[9] 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.

[10] D. Plaisted and A. Sattler-Klein. Proof lengths for equational completion. *Information and Computation*, 125:154–170, 1996.

[11] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 21(7):583–585, 1984.

[12] W. Snyder. A fast algorithm for generating reduced ground rewriting systems from a set of ground equations. *Journal of Symbolic Computation*, 15(7), 1993.

[13] A. Tiwari, L. Bachmair, and H. Ruess. Rigid E-unification revisited. In D. McAllester, editor, *17th Intl Conf on Automated Deduction, CADE-17*, 2000.