Shostak's Congruence Closure as Completion

Deepak Kapur*

Department of Computer Science
State University of New York
Albany, NY 12222
kapur@cs.albany.edu

Abstract. Shostak's congruence closure algorithm is demystified, using the framework of ground completion on (possibly nonterminating, non-reduced) rewrite rules. In particular, the canonical rewriting relation induced by the algorithm on ground terms by a given set of ground equations is precisely constructed. The main idea is to extend the signature of the original input to include new constant symbols for nonconstant subterms appearing in the input. A byproduct of this approach is (i) an algorithm for associating a confluent rewriting system with possibly nonterminating ground rewrite rules, and (ii) a new quadratic algorithm for computing a canonical rewriting system from ground equations.

1 Introduction

Equality reasoning has been found critical in many applications including compiler optimization, functional languages, and reasoning about data bases, most importantly, reasoning about different aspects of software and hardware — circuits, programs and specifications. Significance of congruence closure algorithms on ground equations in compiler optimization and verification applications have long been recognized. Particularly, in the mid 70's and early 80's, a number of algorithms for computing congruence closure were reported in the literature [9, 3, 14]. Congruence closure has been used as a glue to tie different decision procedure for various theories arising in the application of verification and specification analyses. Two related but different approaches are discussed in [15, 11, 12]. These approaches have been implemented in verification systems, e.g., [1, 2]. In our theorem prover *Rewrite Rule Laboratory* (*RRL*) [8], congruence closure is used to implement *contextual rewriting* [17], which combines equality reasoning using rewrite rules with decision procedures for booleans, arithmetic and freely constructed data structures. This tight integration of rewriting with decision procedures by the mechanism of contextual rewriting is perhaps a main reason for the success of *RRL* in automatically verifying arithmetic circuits such as multipliers and adders [6, 7].

As the importance of decision procedures in verification systems is being recognized, interest in algorithms for combining decision procedures is reviving.

* Partially supported by the National Science Foundation Grant nos. CCR-9308016, and CCR-9404930.

Particularly, in a recent paper [2], Cyrluk et. al. resurrected Shostak's framework for combining decision procedures, and argued that this framework is better from efficiency considerations than the one proposed by Nelson and Oppen based on propagation of ground equations. Shostak's congruence closure algorithm serves as the core of his combination framework [14, 15]. Decision procedures satisfying certain properties can be tightly integrated with the basic congruence closure algorithm to give a combination of decision procedures. In contrast, Nelson and Oppen's approach [11, 12] relies on decision procedures working independently and deriving ground equations, which are passed to the congruence closure algorithm. New equations deduced by the congruence closure are, in turn, passed back to the decision procedures. The interaction/cooperation among decision procedures is through the congruence closure and is not considered as tight as in Shostak's framework. According to [2], this results in considerable redundancy in Nelson and Oppen's combination procedure, and the same ground equations being deduced repeatedly.

Another interesting distinction between Shostak's method and Nelson and Oppen's method is that the term universe under consideration should be known a priori in Nelson and Oppen's method, whereas this is not necessary for Shostak's algorithm. Shostak's algorithm computes canonical forms for ground terms with respect to the ground equations already processed. The algorithm implicitly computes a canonical rewriting system represented using its data structures, but what that canonical rewriting system is, is unclear from the original description of the algorithm or the discussion in [2].

In this paper, the canonical rewriting system implicitly computed by Shostak's congruence closure algorithm is made explicit. It is shown that Shostak's congruence closure algorithm can be easily couched in a rewriting framework, thus providing a new insight and better understanding. In fact, Shostak's algorithm allows ground equations to be used in the direction preferred by the user. This translates to relaxing the requirement that the rewrite rules be terminating, i.e., they could be non-terminating. The requirement that rewriting rules be terminating has almost always been considered essential for generating a canonical system using completion [16].

A new method for generating a confluent rewriting system from nonterminating rewrite rules is a byproduct of this connection. We are not aware of any work on generating a confluent rewriting system from nonterminating rules other than a mention in [16] that Snyder's "first algorithm does not depend upon the ordering." The proposed reformulation of Shostak's algorithm also gives a new quadratic method ($O(n^2)$, where n is the size of the ground equational system) for generating a canonical ground rewriting system from ground equations when the original signature can be extended. In contrast, in [4, 16], algorithms are given by first constructing the congruence closure and then introducing a distinct new constant for each congruence class.

1.1 Related Work

It has been well-known that the congruence closure of ground equations can be computed using a completion procedure [5]. The main idea is to orient ground equations into terminating rewrite rules, and then normalize (inter-reduce) the rewriting system. Unlike nonground equations, ground equations can always be oriented into terminating rewrite systems. For instance, it is this algorithm which is used for implementing contextual rewriting in RRL.

A number of papers on generating a complete system of ground rewrite rules in polynomial time from ground equations have been published (see [16] for such references and details). These algorithms first use a congruence closure algorithm on the subterm graph (e.g., using the algorithm of Downey, Sethi and Tarjan [3]) generated from ground equations for computing congruence classes. A unique representative from each congruence class is then picked to generate a complete reduced set of rewrite rules. If the signature can be extended by introducing new constants, generation of a complete rewrite system is relatively easy [4, 16]. Until recently, however, it has been quite a challenge to develop a completion algorithm of polynomial complexity for generating a complete rewrite system directly from ground equations without explicitly constructing the congruence closure using a graph-based congruence closure algorithm. Plaisted and Sattler-Klein [13] proposed two polynomial time algorithms.

The proposed work is different in spirit. Firstly, it is an attempt to explain a particular congruence closure algorithm due to Shostak [15] that has been found useful and implemented in theorem proving and verification systems. Since the algorithm computes canonical forms of terms in an incremental fashion, there is a canonical rewrite system implicitly being computed. This paper demystifies this construction. By incrementally extending the signature as input equations are processed, it is shown that a set of rewrite rules can be given for computing canonical forms of terms represented as dags. New constants introduced correspond to the names of subterms of terms appearing on the right side of ground equations and serving as canonical forms.

Secondly, connecting Shostak's algorithm with completion gives (i) a new polynomial-time algorithm for associating a canonical rewrite system with ground equations, as well as (ii) an algorithm for generating a confluent rewrite system from possibly nonterminating ground rewrite rules.

Thirdly, we provide indirectly a complexity analysis of Shostak's algorithm, which, to our knowledge, has not appeared anywhere in the literature.

2 Shostak's Congruence Closure Method

Below, we discuss some of the salient features of Shostak's algorithm, discussing some key data structures. An example is used to illustrate the main steps. For the original description of Shostak's algorithm, see [15] or [2]; the discussion below is based on [2].

The input to Shostak's algorithm is a finite set S of ground equalities, i.e., $S = \{a_i = b_i \mid 1 \leq i \leq n\}$, where the intent is that b_i is to be preferred over a_i to serve as a representative of the congruence class containing a_i and b_i. (In the rewriting framework, the equation should be oriented as $a_i \rightarrow b_i$, irrespective of whether the induced rewriting relation is terminating or not.) The objective is to incrementally generate a function *canon* to serve as a canonicalizer for

S as equations in S are processed. Later, it is discussed how this algorithm can be further optimized if the set S is known a priori, thus giving an off-line (nonincremental) version.

2.1 Shostak's Algorithm reviewed

```
Shostak(S) =
        CASES S OF
           nil : RETURN,
   [a = b, S'] : Merge( canon(a), canon(b));
                 Shostak(S')
        ENDCASES

Merge(a, b) =
    UNLESS a = b DO
          union(a, b);
          FOR u IN use(a) DO
             replace a by b in the argument list of sig(u);
             FOR v In use(b) WHEN sig(v) = sig(u) DO
             Merge(find(u), find(v));
          use(b) := use(b) U {u}

canon(t) =
           CASES t OF
             f(t_1,...,t_n): canonsig(f(canon(t_1),...,canon(t_n))),
             ELSE: canonsig(t)
           ENDCASES

canonsig(t) =
          CASES t OF
            f(t_1,...,t_n):
              IF (FORESOME u IN use(t_1):t=sig(u))
                  THEN RETURN find(u)
                  ELSE FOR i FROM 1 to n DO use(t_i):=use(t_i) U {t};
                    sig(t) := t;
                    use(t) := {};
                    RETURN t
              ENDIF
            ELSE: find(t);
          ENDCASES;
```

There are three inter-twined phases in Shostak's algorithm during which a new equality $s = t$ is processed. The first and perhaps the most important phase is to compute the canonical forms of s and t from ground equations already processed. The function *canon* does this using two data structures *use* and *sig*. If the two canonical forms are not identical, then the equivalence class containing

s is *merged* into the equivalence class containing t; this is typically done using (naive) *union-find* primitives and data structure to avoid overhead. The third phase is to propagate this new equality to compute the congruence closure, by suitably updating use, $find$ and sig data structures, as well as recursively calling $merge$.

The data structure use can be viewed as collecting dependency among all the terms processed so far. For a term t not in canonical form, $use(t)$ gives the set of all terms in which t is a top-level argument; if t is in canonical form, then $use(t)$ gives the set of all terms whose signature contains t as a top-level argument. The data structure sig is a helper to the *find* primitive for computing canonical forms of terms not yet seen by *find*; it maintains the property that $sig(f(t_1, \cdots, t_n)) = f(find(t_1), \cdots, find(t_n))$. These data structures are modified as equalities are processed, and used along with $find$ to compute canonical forms of terms. All the data structures are initialized to be the empty sets.

Canonical forms are computed from inside out. The canonical form of a constant is computed by invoking $find$. The canonical form of a non-constant term is computed by recursively computing the canonical forms of its arguments using $canonsig$. Once a canonical form of a subterm u is computed, then the canonical form of the subterm in which u appears as an argument, is computed using the data structures use and sig.

Example 1: As a simple example, consider $f(a) = g(f(a))$. The term $f(a)$ is processed for computing its canonical form with respect to the empty set of equations. The canonical form of a is computed first, which is a itself; for computing the canonical form of $f(a)$, then it is checked first whether $f(a)$ already appears among subterms in which a appears as an argument using the data structure $use(a)$. Since it does not, $use(a)$ is updated to include $f(a)$; $sig(f(a))$ is also updated to be $f(a)$, and $f(a)$ itself is the canonical form of $f(a)$. To compute the canonical form of $g(f(a))$, the canonical form of a is computed which is a; then, the canonical form of $f(a)$ is computed which is $find(f(a)) = f(a)$ since $use(a)$ now includes $f(a)$ and $f(a) = sig(f(a))$. Finally, while generating the canonical form of $g(f(a))$, $use(f(a))$ is updated to be $\{g(f(a))\}$, $sig(g(f(a)))$ is made $g(f(a))$; the result is $g(f(a))$.

Now that use, sig data structures are in place, the equality is processed first by merging the congruence classes, which first leads to applying $union$ on $f(a)$ and $g(f(a))$, and then replacing $f(a)$ by $g(f(a))$ in the sig of every subterm appearing in $use(f(a))$. I.e., since $use(f(a)) = \{g(f(a))\}$, $f(a)$ in $sig(g(f(a))) = g(f(a))$ is replaced by $g(f(a))$ to give $g(g(f(a)))$. $use(g(f(a)))$ is made to be $g(f(a))$. So after processing the equality we have:

$use(a) = \{f(a)\},\ \ sig(a) = a,\ find(a) = a,$

$use(f(a)) = \{g(f(a))\},\ \ sig(f(a)) = f(a),\ find(f(a)) = g(f(a)),$

$use(g(f(a))) = \{g(f(a))\},\ \ sig(g(f(a))) = g(g(f(a))),\ find(g(f(a))) = g(f(a)).$

With these data structures in place, the canonical forms can be computed:

$canon(a) = a,\ \ canon(f(a)) = g(f(a)),\ \ canon(g(a)) = g(a),$

$canon(g(f(a)) = g(f(a)),\ \ canon(f(f(a))) = f(g(f(a))),$

$canon(f(g(f(a)))) = f(g(f(a))),$

and so on.

Particular attention should be paid to the canonical form of $f(a)$, which is $g(f(a))$. In other words, the canonical form $g(f(a))$ includes a proper subterm that is not in canonical form. Also note that the canonical form of $g(f(a))$ is $g(f(a))$ itself even though $f(a)$ in $g(f(a))$ is not in canonical form. However, the canonical form of $f(f(a))$ is $f(g(f(a)))$ implying that the subterm $f(a)$ in $f(f(a))$ has to be replaced by its canonical form.

The reader is invited to guess a canonical rewriting system for the above equation that would generate the above canonical forms as computed by Shostak's algorithm. It was this perplexing puzzle after reading [2] which led us to develop the approach proposed in this paper.

3 Reinterpreting Shostak's Algorithm as Completion

In this section, we demystify the canonical rewriting system generated by Shostak's algorithm. We rephrase Shostak's algorithm as building and manipulating rewrite rules so that computing a normal form using the rewriting system generates canonical forms generated by Shostak's algorithm.

The main idea is to extend the signature by introducing new constants for nonconstant subterms appearing in ground equations, and manipulating rewrite rules in the extended signature. This is equivalent to incrementally building a *subterm graph,* which is a directed acyclic graph (dag) (as in [16, 11, 4]), introducing names for pointers to nonconstant subterms, and rewriting the subterm graph using these rewrite rules. A *table* mapping the new constants to the subterms they represent in the original signature is also maintained. This table is used at the end to map the canonical forms in the extended signature to the canonical forms in the original signature. The pseudo-code for the new algorithm is given below.

There are two steps in the completion-based view of Shostak's algorithm. In the first step, a given equation is processed by replacing nonconstant subterms by new constants. First, the equation is normalized by the rewriting system already built. If the two sides in the normalized equation are not equal, a new rule must be made. There are two cases:

1. Both sides are constants: The equation is oriented into a rewrite rule using the termination ordering built so far or by extending the termination ordering so that the equation is oriented from left to right.
2. At least one side is not a constant: New constants are introduced for new subterms, processing them inside out. These definitions of new constants as subterms in the original signature are recorded in $Table$. A rewrite rule for each new constant is included in R_1, with the new constant as the right side of the rewrite rule; this is done by making every new constant smaller than nonconstant symbols in the original signature in the termination ordering. Once all subterms have been replaced by new constants, a new rule relating two constants is made from the equation by attempting to orient it from left to right by appropriately extending the termination ordering on the extended signature.

The function *Make_Rule* does this by calling the *Process* function, which (i) introduces new constants for a nonconstant term, (ii) updates *Table* definitions, and (iii) adds new constant-introducing rules in R_1. It thus modifies *Table*, R_1 and R_2 below, which are global data structures. It returns a constant corresponding to its argument (which is the same as the canonical form of its input). If the input to *Process* is a constant, then it returns that constant itself.

```
ShostakR(S) =
                  Table := {};
                  CASES S OF
                      nil : RETURN,
              [a = b, S'] : ca := canon_e(a); cb := canon_e(b);
                      If not(ca = cb) then Make_Rule(ca, cb);
                            ShostakR(S')
                  ENDCASES

Make_Rule(a, b) =
                ca := Process(a);
                cb : = Process(b);
                Extend(>, <ca, cb>);
                r := { ca -> cb } (or { cb -> ca});
                Inter-reduce(R1 U R2, r);

Process(a) =
         For each nonconstant subterm (enumerated inside-out) s of a,
             c := newconstant();
             R1 := R1 U {s -> c};
             Table := Table U { c = replace(s, Table)};
         return c introduced for a.

canon_e(t) = normalform(t, R1 U R2).

canon(t) = replace(canon_e(t), Table).
```

In the termination ordering, new constants are made lower in precedence than the nonconstant symbols in the original signature. As for constant symbols in the original signature, their precedence in relation to the new constants is determined as equations are processed. The termination ordering is extended so that an equation can always be oriented from left to right. This is how user's preference for the right side in an equation as possibly serving the canonical form of its left side is imposed.

Rules for introducing new constants, henceforth called *constant-introducing* rules, are included in the first part, denoted by R_1, of the rewrite system. The rule corresponding to the equation is oriented and included in the second part, denoted by R_2, of the rewrite system.[2]

[2] Decomposition of the rewriting system into two parts is not essential for the algo-

As an example, consider the processing of $g(f(f(a,b),b))$ by *Process*. First a constant introducing rule $f(a,b) \rightarrow C_1$ is added to R_1, with *Table* including the definition $C_1 = f(a,b)$; another constant introducing rule $f(C_1,b) \rightarrow C_2$ is added to R_1 with *Table* including the definition $C_2 = f(f(a,b),b)$. Finally, $g(C_2) \rightarrow C_3$ is added to R_1 with *Table* including the definition $C_3 = g(f(f(a,b),b))$. Then, *Process* returns C_3.

Proposition 1. *Every rule added to R_1 is of the form $g \rightarrow C$, where g is a ground term $f(C_1,\cdots,C_k)$, and $C, C_1,\cdots,C_k$ are constants.*

Proposition 2. *Every rule added to R_2 is of the form $C_i \rightarrow C_j$, where C_i and C_j are constants.*

Adding a constant introducing rule into R_1 does not simplify any existing rules in $R_1 \cup R_2$ since its left side is a term not seen before. Adding a rule into R_2 can however simplify rules in R_1 as well as R_2. Such a rule replaces one constant by another, which can be done efficiently by maintaining back-pointers from constants to subterms (i.e., back-pointers from subterms to terms containing the subterms, as in the data structure *use* in Shostak's algorithm).

1. A rule in R_1 as well as R_2 becomes redundant if its two sides become equal or it becomes identical to another rule.
2. A rule in R_2 may have to be replaced by an equation if after constant replacement, its left side becomes smaller than its right side. Since the equation obtained by normalization of the rule includes constants already known before, its reorientation can be done without causing any nontermination.
3. The left side of a rule in R_1 may become equal to the left side of another rule in R_1. This will result in an equation relating two distinct constants, which must be oriented using the termination ordering.

The above is done in the function `inter-reduce`, which works the same way as `inter-reduce` in a completion procedure such as the Knuth-Bendix procedure. It takes a rule set $R = R_1 \cup R_2$ and a new rule r as its input, and reduces the rules in R by r. It processes every rule in R_1 first, and then rules in R_2. If any rule in R_1 becomes redundant, it is deleted. If a rule in R_1 reduces to an equation relating two constants (this would happen if the left sides of two rules in R_1 become identical), it is oriented and added to R_2, deleting the rule in R_1. This new rule is then recursively used for inter-reduction. Rules in R_2 are deleted if they become redundant; otherwise some rules may have to be reoriented and used for inter-reduction. The order in which new rules (relating two distinct constants) are processed is done by always starting an unprocessed rule with the least right side in the termination ordering.

---

rithm. This is done in the presentation to (i) keep the constant rules separate from nonconstant rules, and (ii) emphasize the distinction between rewrite rules obtained directly from the equations and the rewrite rules for identifying subterms as well as propagation of equations.

Since rules in R_1 and R_2 are always reduced, $R_1 \cup R_2$ is a reduced, canonical rewrite system over the extended signature. The function $canon\_e(t)$ gives the canonical form of t with respect to $R_1 \cup R_2$; the canonical form uses the extended signature. To compute a canonical form of a term in the original signature, new constants in the result in the extended signature generated by $canon\_e$ are replaced by the subterms they represent from the Table.

If $canon\_e(a)$ and $canon\_e(b)$ are identical, then $a = b$ is in the congruence closure of the equations corresponding to $R_1 \cup R_2$. In that case, no rule corresponding to $a = b$ has to be included into R_1 or R_2.

The above description is somewhat complicated because of the incremental nature of the algorithm. For the nonincremental version, the reader may consult subsection 3.3.

To check whether an equation $s = t$ is in the congruence closure of the set S of ground equations, the canonical forms of s and t are computed using $canon\_e$ using $R_1 \cup R_2$ constructed from S; if they are the identical, then $s = t$ is in the congruence closure of S; otherwise it is not.

3.1 Examples

We illustrate the above procedure by the above example used for illustrating Shostak's algorithm.

Example 1 Contd.: Consider the equation set

$$S_1 = \{f(a) = g(f(a))\}.$$

A new constant C_1 is introduced for $f(a)$. A constant-introducing rewrite rule $f(a) \to C_1$ is defined. The definition $C_1 = f(a)$ is included in Table. This rewrite rule reduces $f(a)$ to C_1 in the equality, thus giving $C_1 = g(C_1)$. A new constant is introduced for $g(C_1)$ using a rewrite rule $g(C_1) \to C_2$. The definition $C_2 = g(f(a))$ is included in Table (it does not matter whether the definition $C_2 = g(C_1)$ or $C_2 = g(f(a))$ is stored in Table). The new rewrite rule rewrites the equality to $C_1 = C_2$. In order to orient the rules in R_1 so they are terminating, the precedence relation $f > C_1$ and $g > C_2$ is used.

A rewrite rule $C_1 \to C_2$ is made from this equality by extending the ordering by $C_1 > C_2$. This rewrite rule simplifies the constant introducing rules, producing the final set of rewrite rules in the canonicalizer to be:

$$R_1 = \{f(a) \to C_2, \ \ g(C_2) \to C_2\}, \ \ R_2 = \{C_1 \to C_2\}.$$

Any rule in R_2 whose left side is a new constant can be deleted from R_2 at the end. This is so since (i) R_1 is reduced so does not have any occurrence of such a constant, (ii) the canonical form computation of a term in the original signature cannot result in intermediate terms with that constant.

To generate canonical forms, we use $R_1 \cup R_2$ and then to translate back to the original signature, constant definitions from Table are used. For example, the canonical form of $f(a)$ is $g(f(a))$ obtained by rewriting $f(a)$ to C_1 which further rewrites to C_2. Then replacing C_2 by its definition gives $f(g(a))$. Similarly, the canonical form of $g(f(a))$ is C_2, which translates to the original signature to give $g(f(a))$. The canonical form of $g(g(f(a)))$ is also C_2. This implies that $g(f(a)) = f(a) = g(g(f(a)))$.

What about the canonical form of $f(f(a))$? $f(f(a))$ simplifies to $f(C_2)$ which is $f(g(f(a)))$. And, the canonical form of $g(a)$ is $g(a)$.

On the original signature, the rewriting system induced is:

$$\{f(a) \rightarrow g(f(a)), \;\; g(g(f(a))) \rightarrow g(f(a))\}.$$

This is obtained by replacing the definitions of the new constants from Table, and discarding any duplications.

The reader should notice that the above rewriting system is nonterminating as well as is not reduced, because the right side of the first rule includes its left side. For generating normal forms, it should be understood that $f(a)$ appearing as an argument to g is already in normal form. That is why the need for the second rule, as otherwise, without the second rule, $g(g(f(a)))$ would be in normal form also.

Example 2: Consider

$$S_2 = \{a = f(a), a = g(a)\}.$$

Processing $f(a)$ involves introducing a new constant C_1 using the rewrite rule $f(a) \rightarrow C_1$ in R_1 and then a rule $a \rightarrow C_1$ is included in R_2, which simplifies the rule in R_1 to $f(C_1) \rightarrow C_1$. *Table* includes $C_1 = f(a)$. The ordering used is $f > C_1$ and $a > C_1$.

Processing the second equation $a = g(a)$ gives $C_1 = g(C_1)$ as the canonical forms of a and $g(a)$. A new constant C_2 is introduced for $g(C_1)$ and the rewrite rule $g(C_1) \rightarrow C_2$ is included in R_1. The ordering is extended to include $g > C_2$. *Table* includes also $C_2 = g(f(a))$. A rule $C_1 \rightarrow C_2$ is introduced for the equation into R_2. The ordering is extended to include $C_1 > C_2$. This rule simplifies other rules to give

$$R_1 = \{f(C_2) \rightarrow C_2, \;\; g(C_2) \rightarrow C_2\}, \;\; R_2 = \{a \rightarrow C_2, C_1 \rightarrow C_2\},$$

as a canonical reduced rewrite system. *Table* consists of $C_1 = f(a), C_2 = g(f(a))$. Again the rule $C_1 \rightarrow C_2$ can be deleted from R_2.

The canonical form of $a, f(a), g(a)$ is C_2, which after translation, is $g(f(a))$. The canonical form of $f(f(a))$, $f(g(a))$ and $g(f(a))$ is also C_2. As a matter of fact, all terms have $g(f(a))$ as the canonical form.

The rewriting system induced on the original signature is:

$$\{a \rightarrow g(f(a)), \;\; g(g(f(a))) \rightarrow g(f(a)), \;\; f(g(f(a))) \rightarrow g(f(a))\}.$$

3.2 Correctness

We first argue that the congruence closure of the input S of ground equations is precisely the congruence closure of the equation sets corresponding to the rewrite system $R_1 \cup R_2$ and the definitions in *Table* when restricted to the original signature. This follows from the observations that

1. $canon\_e(s) = s$ is in the congruence closure of $R_1 \cup R_2$,
2. at every step in `ShostakR`, for a given equation $g_1 = g_2$,
   (a) a rule is added to $R_1 \cup R_2$ if and only if $canon\_e(g_1)$ and $canon\_e(g_2)$ are not identical, and
   (b) rules added to R_1 and R_2 and the definitions added to $Table$ induce the same congruence relation on the original signature as the congruence relation induced by $canon\_e(g_1) = canon\_e(g_2)$ (or $g_1 = g_2$).
3. inter-reduction of $R_1 \cup R_2$ by r in `Make_Rule` does not change the congruence relation induced by $R_1 \cup R_2 \cup \{r\}$.

Theorem 3. *Let $R = R_1 \cup R_2$ be a rewriting system on an extended signature, with Table defining new constants; let R' be obtained by replacing new constants in R by definitions in Table. If R is confluent, then R' is confluent on ground terms in the original signature.*

Proof. Consider g_1, g_2, g_3 over the original signature such that $g_1 \rightarrow^* g_2$ as well as $g_1 \rightarrow^* g_3$ by R'. Since every rule in R' is obtained from a rule in R by replacing new constants by their definitions, every rewrite step in $g_1 \rightarrow^* g_2$ as well as $g_1 \rightarrow^* g_3$ can be mimicked by replacing subterms in g_1, g_2, g_3 by new constants and applying the rewrite rules from R. Let g_i', $1 \leq i \leq 3$, be the terms obtained from g_i by judiciously replacing subterms by new constants so that the rewriting steps can be mimicked. Since R is confluent, g_2' and g_3' are joinable; we can apply the same rewriting steps on g_2 and g_3 respectively from R', thus making g_2 and g_3 joinable. This implies that R' is confluent. □

As, the reader would have noticed, even if R is terminating, R' need not be terminating.

The argument to show that the canonical forms generated by the rewrite system constructed by the proposed algorithm are the same as those constructed by Shostak's algorithm are informal, as there does not exist any characterization (or for that matter, any proof of correctness) of Shostak's algorithm.

3.3 Incrementality vs Off-line

The above algorithm processes ground equations one at a time. An nonincremental version of Shostak's algorithm first processes all the subterms appearing in the set of ground equations given as input. Merging of terms based on equations is done afterwards. This avoids computing normal forms, and produces smaller canonical forms. As will be illustrated below, the canonical forms generated by the off-line algorithm can be different from the canonical forms generated by the incremental version discussed earlier. Further, the algorithm is simpler.

New constants are introduced first for all nonconstant subterms by including constant-introducing rules in R_1. This is equivalent to first building a subterm graph (as in [16, 11, 4]). Propagation of equalities is done by processing each equation, which now relates two constants. For each such equation relating two distinct constants, a rule is added which inter-reduces $R_1 \cup R_2$ as above.

The major gain is that unlike in the incremental version, intermediate normal form computations are avoided.

The algorithm is illustrated on **Example 2** discussed above.

Given the equation set $S_2 = \{a = f(a),\ a = g(a)\}$, two rules introducing constants $\{f(a) \rightarrow C_1,\ g(a) \rightarrow C_2\}$ are included in R_1. Each equation is now processed. The first equation is $a = C_1$, so the rewrite rule $a \rightarrow C_1$ is added to R_2. This rewrite rule normalizes rules in R_1 to give:

$$\{f(C_1) \rightarrow C_1,\ \ g(C_1) \rightarrow C_2\}.$$

Now the second equation $a = C_2$ is processed. First the canonical forms of a and C_2 are computed which gives $C_1 = C_2$. The rewrite rule $C_1 \rightarrow C_2$ is added to R_2. This rule inter-reduces rules in R_1 and R_2 to produce:

$$R_1 = \{f(C_2) \rightarrow C_2,\ \ g(C_2) \rightarrow C_2\}\ \ R_2 = \{a \rightarrow C_2,\ \ C_1 \rightarrow C_2\}.$$

The above is a canonical rewrite system for the congruence closure. Again, the rule $C_1 \rightarrow C_2$ can be deleted from R_2 for the same reason as before. $Table$ consists of $C_1 = f(a),\ C_2 = g(a)$.

Let us compute canonical forms of terms and compare them with canonical forms generated using the incremental version of the algorithm discussed earlier to see if there is any difference. The canonical form of $a, f(a), g(a)$ is C_2, which after translation, is $g(a)$. All terms have $g(a)$ as the canonical form. The canonical form generated using the incremental version of the algorithm is, however, $g(f(a))$.

The rewriting system induced on the original signature is:

$$\{a \rightarrow g(a),\ f(g(a)) \rightarrow g(a),\ g(g(a)) \rightarrow g(a)\}.$$

Example 3: Consider another example due to Snyder. The equation set is:

$$S_3 = \{a = f(f(a)),\ \ a = b\}.$$

The subterms lead to the following rule set

$$R_1 = \{f(a) \rightarrow C_1,\ \ f(C_1) \rightarrow C_2\}.$$

Processing the first equation, which becomes $a = C_2$ leads to adding the rule $a \rightarrow C_2$ to R_2. This rule inter-normalizes R_1 to give:

$$R_1 = \{f(C_2) \rightarrow C_1,\ \ f(C_1) \rightarrow C_2\},\ \ R_2 = \{a \rightarrow C_2\}.$$

Processing the equation $a = b$ which normalizes to $C_2 = b$ adds the rule $C_2 \rightarrow b$ into R_2. This rule inter-normalizes R_1 and R_2 to produce:

$$R_1 = \{f(b) \rightarrow C_1,\ \ f(C_1) \rightarrow b\},\ \ R_2 = \{a \rightarrow b, C_2 \rightarrow b\}.$$

$Table$ consists of $C_1 = f(a),\ C_2 = f(f(a))$.

Using the above rules, the canonical forms for various terms are:

$canon(a) = b,\ canon(f(a)) = f(a),\ \ canon(f(f(a))) = b.$

The rewriting system induced on the original signature is:

$$\{f(b) \rightarrow f(a),\ f(f(a)) \rightarrow b\},$$

since the rule corresponding to $C_2 \rightarrow b$ does not have to be included.

If the above equations are processed in reverse order, i.e., $a = b$ is processed before $a = f(f(a))$, different canonical forms are generated, as is to be expected.

$$R_1 = \{f(a) \rightarrow C_1,\ \ f(C_1) \rightarrow C_2\}.$$

Processing $a = b$ leads to the rule $a \rightarrow b$ to R_2 which reduces the first rule in R_1.

$$R_1 = \{f(b) \rightarrow C_1,\ \ f(C_1) \rightarrow C_2\},\ \ R_2 = \{a \rightarrow b\}.$$

Processing $a = f(f(a))$ leads to the equation $b = C_2$ which is oriented as a rule $b \rightarrow C_2$ and added to R_2. This rule reduces rules in $R_1 \cup R_2$ to give:

$$R_1 = \{f(C_2) \rightarrow C_1,\ \ f(C_1) \rightarrow C_2\},\ \ R_2 = \{a \rightarrow C_2,\ b \rightarrow C_2\}.$$

The canonical forms for various terms are:
$canon(a) = f(f(a))$, $canon(f(a)) = f(a)$, $canon(b) = f(f(a))$,
$canon(f(f(a))) = f(f(a))$, $canon(f(f(f(a)))) = f(a)$.

The rewriting system induced on the original signature is:

$$\{a \rightarrow f(f(a)),\ b \rightarrow f(f(a)),\ f(f(f(a))) \rightarrow f(a)\}.$$

4 Complexity

For complexity analysis of the incremental as well as off-line versions of the above rewrite rule based algorithm, the following steps need to be analyzed:

1. computation of canonical forms,
2. adding constant-introducing rules into R_1, and
3. adding a rule in R_2 which leads to recursively inter-reducing rules in $R_1 \cup R_2$.

Let $|S|$ be the sum of the sizes (number of symbols) of the left and right sides of equations in S. Let $|R_i|$ be the sum of the sizes of the left side and right side of the rules in $R_i, i = 1, 2$. Checking whether a ground equation $s = t$ is in the congruence closure built so far, the canonical forms (in the extended signature) of s and t are computed. As shown above, the right side of every rule in $R_1 \cup R_2$ is a constant; the left side of a rule in R_1 is a constant or a term $f(C_1, \cdots, C_k)$, where the C_i's are constants; the left side of a rule in R_2 is always a constant.

Since the rule sets are reduced, computing the canonical form of a constant takes constant time. The canonical form of a nonconstant term t can be computed in $O(|t| + |R_1| + |R_2|)$ steps ($O(|t|)$ using hashing), since every rewrite step either reduces the size of t or rewrites a constant in it. It can thus be decided whether $< s, t >$ is in the congruence closure of S, in $O(|s| + |t| + |S|)$ steps.

For $s = t$, adding constant-introducing rules to R_1 takes $O(|s| + |t|)$ steps.

The most interesting phase is that of recursive inter-reduction, which is analyzed below.

Proposition 4. *A rule $C_i \rightarrow C_j$ can be used to rewrite rules in $R_1 \cup R_2$ in $O(|R_1 \cup R_2|)$ steps.*

Proof. Since at worst, all rules in $R_1 \cup R_2$ may have C_i occurring in them, a rule $C_i \rightarrow C_j$ would need to rewrite C_i in every rule by C_j. Using back-pointers, this can be done in $O(|R_1 \cup R_2|)$. □

Proposition 5. *Checking rules for duplication and rules with identical left sides in R_1 can be done in $O(|R_1|)$.*

Proof. To check whether two rules are the same (or have identical left sides), takes $O(|l| + |r| + |R_1|)$ steps for a rule $l \rightarrow r$. □

Proposition 6. *Deleting trivial constant rules and reorienting constant rules in R_2 as well as in R_1 (because of identical left sides) can be done in constant time.*

Proof. Deletion can be done in constant time once it is known which rules have to be deleted. This can be found using the previous proposition. Reorientation can be done in constant time. □.

From the above three propositions and the fact that in the worst case, inter-reduction may cause $O(|S|)$ pairs of constants to become equal, it follows that

Lemma 7. *A rule $C_i \rightarrow C_j$ inter-reduces $R_1 \cup R_2$ in $O(|R_1 \cup R_2|^2)$.*

Proof. An inter-reduction involves rewriting $R_1 \cup R_2$ by $C_i \rightarrow C_j$, identifying duplicate rules for deletion and rules with identical left sides for reorientation, which may lead to inter-reducing the rule sets with another rule $C_k \rightarrow C_l$. Since there can be at most $O(|R_1 \cup R_2|)$ such rules (both C_k, C_l are in normal form), *inter-reduce* would result in a reduced canonical system in $O(|R_1 \cup R_2|^2)$ steps. □

For generating a canonical rewrite system, rules of the form $C_i \rightarrow C_j$ are used for inter-reduction. In the worst case, there can be at most $O(|R_1 \cup R_2|)$ such rules, Further, such a rule can be used for inter-reduction at most once. The worst case of using such a rule for inter-reduction is also the total work required to get the canonical rewrite system. Thus, we have:

Theorem 8. *A canonical rewrite system $R_1 \cup R_2$ can be obtained from S in $O(|R_1 \cup R_2|^2)$.*

Theorem 9. *The congruence closure of S can be computed using* `ShostakR` *in $O(|S|^2)$.*

A canonical rewrite system can be constructed from ground equations in $O(|S|^2)$ using the above algorithm, which is the same order of complexity as the second algorithm given in [13].

To check whether $s = t$ is in the congruence closure of S, it takes $O(|s| + |t| + |S|)$ steps in addition to $O(|S|^2)$ steps needed for constructing the congruence closure from input equations. To compute the canonical form of a term s in the original signature takes $O(|s| + |S|)$ since replacing the new constants by their definitions in $Table$ takes $O(|s| + |S|)$.

Acknowledgements: I would like to thank David Cyrluk for clarifying different aspects of Shostak's algorithm, Paliath Narendran for encouragement and patiently listening to my half-baked ideas, and Wayne Snyder, Chris Lynch and Ashish Tiwari for valuable comments on an earlier draft of this paper.

References

1. D. Craigen, S. Kromodimoelijo, I. Meisels, W. Pase, and M. Saaltink, "Eves system description," Proc. *Automated Deduction - CADE 11,* LNAI 607 (ed. Kapur), Springer Verlag (1992), 771-775.
2. D. Cyrluk, P. Lincoln, and N. Shankar, "On Shostak's decision procedures for combination of theories," Proc. *Automated Deduction - CADE 13,* LNAI 1104 (eds. McRobbie and Slaney), Springer Verlag (1996), 463-477.
3. P.J. Downey, R. Sethi, and R.E. Tarjan, "Variations on the common subexpression problem," *JACM,* 27(4) (1980), 758-771,
4. Z. Fülöp, and S. Vagvölgyi, "Ground term rewriting rules for the word problem of ground term equations," *Bulletin of the EATCS,* 45 (1991), 186-201.
5. G. Huet and D. Lankford, On the Uniform Halting Problem for Term Rewriting Systems. INRIA Report 283, March 1978.
6. D. Kapur and M. Subramaniam, "New uses of linear arithmetic in automated theorem proving for induction," *J. Automated Reasoning,* 16(1-2) (1996), 39-78
7. D. Kapur and M. Subramaniam, "Mechanically verifying a family of multiplier circuits," Proc. *Computer Aided Verification (CAV),* New Jersey, Springer LNCS 1102 (eds. R. Alur and T.A. Henzinger) (1996), 135-146
8. D. Kapur and H. Zhang, "An overview of Rewrite Rule Laboratory (RRL), " *Computers and Math. with Applications,* 29(2) (1995), 91-114.
9. D. Kozen, *Complexity of Finitely Presented Algebras.* Technical Report TR 76-294, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1976.
10. D. Knuth and P. Bendix, "Simple word problems in universal algebras," in *Computational Problems in Abstract Algebra* (ed. Leech), Pergamon Press (1970), 263-297.
11. G. Nelson, and D.C. Oppen, "Simplification by cooperating decision procedures," *ACM Tran. on Programming Languages and Systems* 1 (2) (1979) 245-257.
12. G. Nelson, and D.C. Oppen, "Fast decision procedures based on congruence closure," *JACM,* 27(2) (1980), 356-364.
13. D. Plaisted, and A. Sattler-Klein, "Proof lengths for equational completion," *Information and Computation,* 125 (1996), 154-170.
14. R.E. Shostak, "An algorithm for reasoning about equality," *Communications of ACM,* 21(7) (1978), 583-585.
15. R.E. Shostak, "Deciding combination of theories," *Journal of ACM* 31 (1), (1984) 1-12.
16. W. Snyder, "A fast algorithm for generating reduced ground rewriting system from a set of ground equations," *J. Symbolic Computation,* 1992.
17. H. Zhang, "Implementing contextual rewriting," Proc. *Third International Workshop on Conditional Term Rewriting Systems,* Springer LNCS 656 (eds. J. L. Remy and M. Rusinowitch), (1992), 363-377.