Grammar Rewriting

David McAllester

MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge Mass. 02139
dam@ai.mi.edu

Abstract. We present a term rewriting procedure based on congruence closure that can be used with arbitrary equational theories. This procedure is motivated by the pragmatic need to handle equational theories where confluence can not be achieved. The procedure uses context free grammars to represent equivalence classes of terms. The procedure rewrites grammars rather than terms and uses congruence closure to maintain certain congruence properties of the grammar. Grammars provide concise representations of large term sets. Infinite term sets can be represented with finite grammars and exponentially large term sets can be represented with linear sized grammars. Although the procedure is primarily intended for use in nonconfluent theories, it also provides a new kind of confluence that can be used to give canonical rewriting systems for theories that are difficult to handle in other ways.

This paper appeared in CADE-11, 1992.
A postscript electronic version of this paper can be found in internet file ftp.ai.mit.edu:/pub/dam/cade92.ps
A bibtex reference for this paper can be found in ftp.ai.mit.edu:/pub/dam/dam.bib

1 Introduction

In most practical applications of term rewriting systems, such as verifications done with the Boyer-Moore prover [3], confluence can not be achieved. In such cases attempts to prove true equations often fail. In this paper a new term rewriting procedure is described that is intended to improve the success rate for proof attempts in nonconfluent theories. In cases where normal term rewriting fails, success can sometimes be achieved by generating a set of normal forms rather than an individual normal form. This is done using a term ordering where a single term can have many normal forms all of which are minimal under that ordering. Somewhat surprisingly, canonicalization under such weak term orderings is possible. The canonical form of a given term is taken to be the *set* of all normal forms.

The main problem with using sets as canonical forms is that the sets involved can be quite large. In fact, to improve the success rate in nonconfluent theories we would like the canonical sets to be as large as possible. Fortunately, large sets can be compactly represented with context free grammars. A finite grammar can represent an infinite set of terms. More importantly, very large finite term sets can be represented by compact grammars. Consider the equational theory consisting of the single commutativity axiom of the form $x + y = y + x$. In this theory the equivalence class of a sum of n constants contains order 2^n terms. However, a grammar for generating this class contains only order n productions. As another example consider the theory consisting of a single associativity axiom $x+(y+z) = (x+y)+z$. In this theory the size of the equivalence class of a sum of n constants is the Catalan number of n — a hyperexponential function. However the equivalence class of a sum of n constants can be generated by a grammar with order n^3 productions. In the equational theory that contains both associativity and commutativity the grammar for generating the equivalence class of a sum of n constants grows exponentially in n. However, the grammar is still vastly smaller than the equivalence class itself.

Using grammars to represent equivalence classes involves the well known congruence closure procedure. Congruence closure is an efficient algorithm for determining the consequences of a finite set of ground equations [8], [13], [11], [5]. A finite set of ground equations can be converted to a grammar that encodes the congruence relation on terms implicit in the equations. Congruence closure can be viewed as an algorithm for converting ground equations to grammars. The rewriting procedure described here operates on grammars — a grammar is repeatedly rewritten to incorporate new ground equations.

The rewriting procedure described here is analogous to ordered rewriting systems [2], [6], [9], [12]. It rewrites (representations of equivalence classes of) ground terms using a set of unordered equations. This rewriting is done in the presence of a well founded order on ground terms and the rewriting process is guaranteed to terminate. However, unlike previous ordered rewriting systems, the order on ground terms is not assumed to be total. The commutativity equation $x + y = y + x$ can be handled by placing *both* sides of the equation into the representation of the set being rewritten. Thus the grammar rewriting procedure described here provides a way of handling non-orientable equations that is different from both ordered rewriting and from the use of special unification algorithms [7].

The grammar rewriting technique described here is somewhat similar to a congruence closure based rewriting technique for nonground rewrite rules described by Chew [4]. However, Chew's procedure requires that the set of rewrite rules be nonoverlapping, i.e., that there are no critical pairs. Chew's procedure also essentially requires that the normal form of a term be a single term rather than a set represented by a grammar.

The full version of this paper (available form the author) shows how grammar rewriting can be used to give a canonical rewriting system for idempotent semigroups. It has been shown that there is no canonical term or word rewriting system for this theory [1]. Unfortunately, the canonical grammar rewriting system for idempotent semigroups is highly exponential and is of theoretical interest only.

The grammar rewriting procedure described here has been incorporated into the Ontic verification system which is under continued development by the author, Robert Givan, Carl Witty, and Kevin Zalondek. Experimentation with the procedure in Ontic is currently under way.

2 Congruence Grammars

This section reviews basic properties of congruence grammars. A congruence grammar is formally equivalent to a deterministic bottom-up finite state tree automaton [14] [10]. The grammar view used here seems more natural than the tree automaton view in the term rewriting domain.

Congruence grammars provide a way of compactly representing equivalence classes of first order terms. Each equivalence class is represented by a nonterminal symbol of the grammar which generates the elements of the class. Equivalence classes are always disjoint sets. This observation motivates the restriction that no two productions of a congruence grammar can have the same right hand side. For example, we can not have $X \rightarrow a$ and $Y \rightarrow a$ where X and Y are distinct nonterminal symbols.

> **Definition:** A *Congruence Grammar* is a set of productions of the form $X \rightarrow f(Y_1, \cdots, Y_n)$, where f is an n-ary function symbol, each Y_i is a nonterminal symbol, and where no two productions have the same right hand side.

We use the standard definition of a first order term where we assume an infinite set of function symbols of each arity (number of arguments). Constant symbols are treated as function symbols of no arguments. So, for example, the production $X \rightarrow a$ is a production of the above form where a is function of no arguments. The above definition allows us to prove that distinct nonterminal symbols generate disjoint classes. The following lemma can be proved by induction on the size of terms.

Lemma: A given term is generated by at most one nonterminal of a congruence grammar.

Example, Representing an Infinite Class: Consider the equivalence class of the constant symbol a under the single equation $a = f(a)$. This infinite equivalence class is a context free language generated by the two productions $X \to f(X)$ and $X \to a$.

Example, An Equivalence Class under Commutativity: Consider n distinct constants $a_1, a_2, \cdots, a_n$ and define the term t_i to be the sum of the first i constants associated to the left, i.e., t_1 is a_1, t_2 is $a_1 + a_2$, t_3 is $(a_1 + a_2) + a_3$, and so on. Note that for $1 < i \leq n$ we have that t_i is $t_{i-1} + a_i$. Suppose that $+$ is commutative but not associative. In this case the equivalence class of the term t_n is generated by the nonterminal X_n in the grammar containing the $2(n-1)$ productions of the form

$$X_i \to (a_i + X_{i-1})$$
$$X_i \to (X_{i-1} + a_i)$$

together with the production

$$X_1 \to a_1.$$

The 2^{n-1} terms in this equivalence class are generated by a grammar with $2n-1$ productions.

Example, An AC Equivalence Class: Consider n constants $a_1, a_2, \cdots, a_n$ and suppose that $+$ is both associative and commutative. Let S be any non-empty subset of these constants. We define t_S to be the term that is the sum of the elements of S. For each non-empty subset S of the constants we introduce a nonterminal symbol X_S. Now consider the grammar which contains all productions of the form

$$X_S \to (X_W + X_V)$$

where W and V are disjoint and S is $W \cup V$ plus all productions of the form $X_{\{a_i\}} \to a_i$. The equivalence class of t_U is generated by the nonterminal X_U of this grammar. More generally, the nonterminal X_S generates the equivalence class of t_S.

The grammar that generates the equivalence class of t_U grows exponentially in size of U — the grammar contains order 3^n productions. This fact can be derived by observing that each production $X_S \to (X_W + X_V)$ classifies each element of U in one of three ways — either as a member of W, or a member of V, or as a member of neither. There are 3^n ways of classifying n constants into three groups. The number of productions in this grammar should be contrasted with the number of terms in the equivalence class. The grammar for the set of terms equivalent to a sum of n variables contains $O(3^n)$ productions while the equivalence class contains $O(4^n(n+1)!)$ different terms.[1] When a term ordering is used the grammar that generates the set of normal forms may contain fewer productions.

[1] The order of the number of terms in the equivalence class is derived from Sterling's approximation applied to a formula involving Catalan numbers.

3 Congruence Grammars and Ground Equations

A congruence grammar is a representation of an equivalence relation on terms. It turns out that those relations which can be represented by finite congruence grammars are exactly those relations which are the deductive closure of a finite set of ground equations. This section states some basic results relating congruence grammars and ground equations.

First, we define the equivalence relation on (all) ground terms represented by a congruence grammar. This is done by defining an interning operation on ground terms. The word "interning" is commonly used to describe the way strings are mapped to variables in most programming languages. Here we are mapping semantically equivalent terms to the same internal data structure. We assume a given congruence grammar and define an interning operation such that for any term t, the result of interning t, denoted $I[t]$, is a nonterminal of the grammar such that $I[t]$ generates t. If there is no nonterminal of the grammar which generates t then the grammar is extended with new productions in such a way that the desired nonterminal is created.

Procedure for Interning a term t:

1. Let $s_1, \cdots, s_n$ be the (possibly empty) sequence of immediate subterms of t, i.e., the terms such that t is $f(s_1, \cdots, s_n)$.
2. Let $Y_1, \cdots, Y_n$ be the result of recursively interning the terms $s_1, \cdots, s_n$.
3. Let $X \rightarrow f(Y_1, \cdots, Y_n)$ be the production whose right hand side is $f(Y_1, \cdots, Y_n)$. If there is no such production then create one with a new nonterminal X and add it to the grammar.
4. Return the nonterminal symbol X.

Since no two productions share the same right hand side the productions can be stored in a hash table indexed by their right hand sides. Throughout this section we assume the presence of such a hash table and assume that hash table operations can be performed in unit time. Under these assumptions the above intern procedure runs in linear time in the size of the given term.

Note that the interning procedure has a "side effect" — it can add productions to the grammar. This ensures that for any sequence of interning operations the nonterminal symbol $I[t]$ that results from interning t will be the same for each repeated interning of t even if the original grammar did not contain any nonterminal that generated t. Although a purely functional treatment of interning is possible, the functional treatment is more complex. In spite of the implementation based on side effects, the intern function is best viewed as a function from terms to nonterminals where the mapping from terms to nonterminals is completely determined by the initial grammar. In fact, the initial congruence grammar determines an equivalence relation on terms such that two terms s and t are equivalent if $I[s]$ equals $I[t]$. For example, suppose the initial grammar contains the two productions $X \rightarrow f(X)$ and $X \rightarrow a$ and let $f^n(a)$ be an abbreviation for the term $f(f(\cdots f(a)))$ with n occurrences of f. In this case we have that $I[f^n(a)]$ equals X for any term of the form $f^n(a)$. This implies that $I[g(f^n(a))]$ will be equal to $I[g(f^m(a))]$ for any nonnegative integers n and

m. Computing $I[g(f^n(a))]$ gives a nonterminal symbol Y such that the grammar has been extended to include the production $Y \rightarrow g(X)$. Although interning can add productions to the grammar, it does not change the equivalence relation on terms defined by the grammar. Different grammars define different intern mappings and we will write $I_G[s]$ to mean the result of interning the term s with respect to the grammar G.

> **Theorem:** For any finite set H of equations between ground terms there exists a finite congruence grammar $G_r(H)$ such that that for any two ground terms s and t we have that $H \models s = t$ if and only if $I_{G_r(H)}[s]$ equals $I_{G_r(H)}[t]$. Furthermore, assuming that hash table operations can be done in unit time, the grammar $G_r(H)$ can be computed from H in $n \log n$ time where n is the written length of H.

The proof of this theorem is based on well known algorithms for congruence closure [8], [13], [11], [5]. A procedure for incrementally incorporating new equations directly into a congruence grammar is given in section 6. A nonoptimal algorithm for converting a set of equations H to a congruence grammar $G_r(H)$ can be described as follows. For each term s occurring in H we introduce a nonterminal symbol X_s.[2] For each constant symbol a appearing in H we construct the production $X_a \rightarrow a$. For each term $f(w_1, \cdots w_k)$ occurring in H we add the production $X_{f(w_1, \cdots w_k)} \rightarrow f(X_{w_1}, \cdots, X_{w_n})$. Now we process the equations in H while maintaining a union-find structure on nonterminal symbols. For each equation $s = w$ in H we call union on the nonterminals X_s and X_w. After processing all equations we select a canonical representative from each equivalence class of nonterminal symbols and replace every nonterminal in the grammar by its canonical representative. The resulting grammar need not be a congruence grammar because it is now possible that two distinct productions have the same right hand side. Any time we have two productions $X \rightarrow f(W_1, \cdots W_n)$ and $Y \rightarrow f(W_1, \cdots W_n)$ with the same right hand side we uniformly replace all occurances of X in the grammar by Y. Such replacement is continued until we have a congruence grammar.

The transformation from equations to grammars has an inverse — one can transform a grammar back into a set of equations. For each nonterminal symbol X we construct a new constant symbol c_X. The constants of the form c_X will be called *internal* constants to distinguish them from the other (external) constants. A term that does not contain any of these internal constants will be called an external term. We now have the following definition and lemma.

> **Definition:** If G is a congruence grammar we define $E_q(G)$ to be a set of equations of the form $c_X = f(c_{Y_1}, \cdots, c_{Y_n})$ where G contains the production $X \rightarrow f(Y_1, \cdots, Y_n)$.

For example, if G contains the two productions $X \rightarrow f(X)$ and $X \rightarrow a$, then $E_q(G)$ contains the two equations $c_X = f(c_X)$ and $c_X = a$.

[2] (We say that s occurs in H if s is either one side of an equation in H *or* s is a subterm of a term in an equation in H.

Lemma: For any congruence grammar G, and any two external ground terms s and t, we have that $I_G[s]$ equals $I_G[t]$ if and only if $E_q(G) \models s = t$.

Proof Sketch: One can show by induction on the size of a term s that $E_q(G) \models s = c_{I_G[s]}$. Then if $I_G[s] = I_G[t] = X$ we have $E_q(G) \models s = c_X$ and $E_q(G) \models t = c_X$ so $E_q(G) \models s = t$. To show the converse we add a production $X \to c_X$ for each nonterminal X. These productions do not alter the intern function on external terms. We then consider the congruence relation (on both internal and external terms) defined by the intern function for this extended grammar. This congruence relation provides a semantic model of $E_q(G)$. So if $I_G[s] \neq I_G[t]$ then $E_q(G) \not\models s = t$. ■

The operation E_q introduces internal constants into the equation set. These internal constants are irrelevant to the equivalence relation on external terms defined by the equation set. One can define the operation G_r so that it eliminates internal constants from the grammar. The equation sets constructed by E_q are useful for analysis and conceptual definitions but they are never actually computed. All computation is done directly on grammars.

4 Grammar Rewriting

This section defines the basic concepts of grammar rewriting.

Definition: A grammar rewriting system is a pair $<E,\ w>$ where E is a set of equations between first order terms (usually containing variables) and w is a weight function which assigns a positive integer to every ground term.

The weight function induces a well founded order on ground terms by setting $s < t$ if and only if $w[s] < w[t]$. Unlike the orderings used in traditional ordered rewriting systems, e.g., [9], the weight orderings used in grammar rewriting are not total on ground terms. In term rewriting one is interesting in simplifying one particular term. Grammar rewriting is also focused on a particular term. However, in grammar rewriting this term is represented by a nonterminal symbol of a grammar.

Definition: We define the "one step" rewrite relation $\mapsto_E$ on ground terms so that $\sigma[s] \mapsto_E \sigma[t]$ provided either $s = t \in E$ or $t = s \in E$, every free variable of t appears in s, and σ is a ground substitution.

For example, if E is the set $\{g(x, f(x)) = c\}$ then we have that $g(h(a),\ f(h(a))) \mapsto_E c$ but we do not have $c \mapsto_E g(h(a),\ f(h(a)))$.

Definition: A *grammar term* is a pair $<X,\ G>$ where G is a finite congruence grammar and X is a nonterminal that appears in G.

Definition: We say that a term s is a minimal representative of a grammar term $<X, G>$, with respect to a weight function w, if s is generated by X under G and no other term generated by X is smaller than s according to w.

Definition: We define the one step rewrite relation $\mapsto_{<E, w>}$ on grammar terms so that

$$<X, G> \mapsto_{<E, w>} <X', G_r(E_q(G) \cup \{u = v\})>$$

provided u is a subterm of some minimal representative of $<X, G>$ under the weight function w, $u \mapsto_E v$, $I_G[u] \neq I_G[v]$, and X' is the nonterminal of $G_r(E_q(G) \cup \{u = v\})$ that generates the terms generated by X under G.

The one-step grammar rewriting operation defined above corresponds to selecting a term minimally generated by X, rewriting that term according to some equation in E, and then modifying the grammar so that the result of the rewrite is included in the language generated by X. A procedure for efficiently computing $G_r(E_q(G) \cup \{w = u\})$ from G and the equation $w = u$ is given in section 6. This procedure does not construct an equation set — it performs a direct operation on the grammar.

The definition of the rewrite relation on grammars requires that the new equation being incorporated into the grammar is indeed new, i.e., it can not be an equation that is already implied by $E_q(G)$. This allows for the existence of normal forms as defined below.

Definition: We say that a grammar term $<X, G>$ is *in normal form* if there is no grammar term $<X', G'>$ such that $<X, G> \mapsto_{<E, w>} <X', G'>$.

As an example of rewriting, let C be the equation set consisting of the single commutative law $x + y = y + x$. Let 1 be the weight function that assigns every term the weight 1 (this weight function corresponds to the empty ordering on terms). Let G be the grammar consisting of the productions $X_1 \rightarrow a_1 + X_2$, $X_2 \rightarrow a_2 + X_3, \cdots, X_{n-1} \rightarrow a_{n-1} + X_n$, $X_n \rightarrow a_n$. In other words, G is the grammar generated by interning the term $a_1 + (a_2 + (\cdots + a_n))$ starting with the empty grammar. In the grammar rewriting system $<C, 1>$ we have that the grammar term $<X_1, G>$ rewrites to a normal form $<X_1, G'>$ where G' is the grammar consisting of the productions of the form

$$X_i \rightarrow a_i + X_{i+1}$$

$$X_i \rightarrow X_{i+1} + a_i$$

together with the production

$$X_n \rightarrow a_n.$$

A similar example can be given for the equation set AC consisting of the associative and commutative laws.
Certain restrictions on the weight function w can be used to ensure that grammar rewriting always terminates.

> **Definition:** A grammar rewriting system $<E,\ w>$ is called *terminating* if there are no infinite rewrite chains of the form
>
> $$<X_1,\ G_1> \mapsto_{<E,\ w>} <X_2,\ G_2> \mapsto_{<E,\ w>} <X_3,\ G_3> \mapsto_{<E,\ w>} \cdots$$

> **Definition:** A weight function w will be called a *polynomial weight function* if for each function symbol f of n arguments there exists a polynomial $P_f(x_1,\ \cdots,\ x_n)$ in n variables with coefficients greater than or equal to 1, where each x_i appears in at least one term, where there is a constant term of at least 1, and such that for any ground terms $s_1,\ \cdots,$ s_n we have $w[f(s_1,\ \cdots,\ s_n)] = P_f(w[s_1],\ \cdots,\ w[s_n])$.

Orderings based on polynomial weight functions are well known in the term rewriting literature. For a given finite set of function and constant symbols, and a given weight k, there are only finitely many terms that can be constructed from those symbols that have weight less than or equal to k. Because grammar rewriting with a finite equational theory can not generate infinitely many constant or function symbols, and because matching is restricted to minimal weight terms, we have the following well foundedness lemma for grammar rewriting.

> **Lemma:** If w is a polynomial weight function and E is any finite set of equations then $<E,\ w>$ is terminating.

We use $\mapsto^*_{<E,\ w>}$ to denote the reflexive transitive closure of the relation $\mapsto_{<E,\ w>}$.

> **Definition:** Let $I_0[s]$ be the grammar term that results from interning s relative to the empty grammar. If $I_0[s] \mapsto^*_{<E,\ w>} <X,\ G>$ and $<X,\ G>$ is in normal form, then $<X,\ G>$ is called a *normal form of* s.

> **Definition:** For any ground term s and set of equations E we define $|s|_E$ to be the set of terms that can be proven to be equal to s using the equations in E.

> **Definition:** A grammar rewriting system $<E,\ w>$ is called *complete* if for each ground term s, and any normal form $<X,\ G>$ of s, every minimal member of $|s|_E$ is generated by X under G.

> **Definition:** A grammar rewriting system is called *canonical* if it is terminating and complete.

A canonical grammar rewriting system $<E,\ w>$ provides a decision procedure for the equational theory E.

Theorem: Let $<E, w>$ be a canonical grammar rewriting system and let s and t be any two ground terms. Let $<X, G>$ and $<X', G'>$ be normal forms for s and t respectively under the rewriting system $<E, w>$. Let G'' be the grammar that encodes all equivalences encoded in G or G', i.e., G'' is $G_r(E_q(G) \cup E_q(G'))$. We have that $E \models s = t$ if and only if $I_{G''}[s] = I_{G''}[t]$.

This lemma follows from the invariant that for any ground term u if $I_0[u] \mapsto^*_{<E, w>} <X, G>$ then X generates u under G and if $<X, G>$ is a normal form of u then X generates all minimal elements of $|u|_E$.

5 Examples

We let w be the simple polynomial weight function such that for every term $f(s_1, \cdots, s_n)$ we have that $w[f(s_1, \cdots, s_n)] = w[s_1] + \cdots + w[s_n] + 1$. Let A and C be the two equation sets $\{x + (y + z) = (x + y) + z\}$ and $\{x + y = y + x\}$ respectively. Let AC be $A \cup C$. One can easily verify that the systems A, C, and AC are all canonical under this ordering.

Let w be any polynomial weight function satisfying $w[s+t] = 2w[s]+w[t]+1$. Under this weight function we have that $w[s+t] < w[t+s]$ provided $w[s] < w[t]$ and we have $w[s + (t + u)] < w[(s + t) + u]$ for any terms s, t, and u. The equation set $AC \cup \{x + (y + z) = y + (x + z)\}$ forms a canonical system where every term normalizes to a grammar term whose minimal representatives are the weight sorted permutations of the addends under a standard parenthesization. If the addends are linearly ordered by weight then there is only one minimal representative.

We now consider Abelian groups. Let w be any order satisfying $w[s + t] = w[s] + w[t] + 1$ and $w[-s] = 2w[s]$. Under this ordering we have that $w[(-s) + (-t)] < w[-(s+t)]$. The equation set $AC \cup \{-(x+y) = (-x)+(-y),\ \ x+(-x) = 0,\ \ x + 0 = x\}$ is a canonical system under this ordering. Every term normalizes to a grammar whose minimal representatives form an AC equivalence class. Refinements of the ordering can give canonical systems which generate smaller grammars.

Let E be $AC \cup \{f(x+x) = 1\}$. This is given in [9] as an example of a theory that can not be handled by ordered term rewriting without special unification procedures. However, this equation set is canonical under grammar rewriting using the simple ordering given in the first example of this section.

The theory of idempotent semigroups is the theory defined by the associativity axiom $x + (y + z) = (x + y) + z$ and the idempotence axiom $x + x = x$. The theory idempotent semigroups has the property that for any finite set of constants the set of terms that can be constructed from those constants and the operation $+$ fall into only a finite number of equivalence classes. Such a theory is called *locally finite*. A theory is called *regular* if every equation has the property that every variable appearing in one side of the equation also appears on the other side of the equation. In the full paper it is shown that any locally finite regular theory is canonical under fair grammar rewriting relative to the empty order on terms. Hence grammar rewriting gives a canonical rewriting system for the theory of idempotent semigroups. It has been shown that there is no canonical term or word rewriting system for the theory of idempotent semigroups [1].

6 Incorporating Equations into Grammars

This section gives an algorithm for computing $G_r(E_q(G) \cup \{s = t\})$ from the grammar G and the equation $s = t$. The grammar $G_r(E_q(G) \cup \{s = t\})$ is directly computed from G by incrementally adding and removing productions. The algorithm given here is quite similar to the congruence closure procedure described in [11]. The algorithm has been reformulated here to operate on grammars and optimized to run in order $n \log n$ time (under the assumption that hash table operations take unit time).[3]

This procedure uses the internal constants of the form c_Z that are associated with the nonterminals of the grammar. The procedure maintains an equivalence relation represented by three sets of equations. First, the procedure maintains a congruence grammar. Second, the procedure maintains a queue of equations of the form $c_Z = c_Y$. Third, an additional set of equations between constants of the form c_Z is maintained in a union-find structure on these constants. The equivalence relation determined by these three sets of equations is maintained as a fixed invariant of the procedure.

As mentioned above, equations between constant symbols can be handled by the well known union-find procedure. Consider a set of equations $a_1 = b_1,\ a_2 = b_2,\ \cdots,\ a_n = b_n$. This set of equations can be represented in a union-find structure by executing $\mathtt{union}(a_1,\ b_1)$, $\mathtt{union}(a_2,\ b_2)$, $\cdots$, $\mathtt{union}(a_n,\ b_n)$. To see if an equation $c = d$ follows from the given equations one computes $\mathtt{find}(c)$ and $\mathtt{find}(d)$. The equation $c = d$ is provable if and only if these two find operations return the same value.

Definition: A constant c_X is said to be *dead* if $\mathtt{find}(c_X)$ is some constant other than c_X. A constant that is not dead is said to be *alive*, i.e., a constant c_Y is alive if $\mathtt{find}(c_Y)$ equals c_Y.

Procedure for computing $G_r(E_q(G) \cup \{s = t\})$:

1. Let Z and W be the nonterminals $I_G[s]$ and $I_G[t]$.
2. Initialize S to be a queue containing the single equation $c_Z = c_W$.
3. While the queue S is not empty do the following.
   (a) Remove an equation $c_Z = c_W$ from the queue S.
   (b) Let c_X be $\mathtt{find}(c_Z)$ and let c_Y be $\mathtt{find}(c_W)$.
   (c) If c_X is the same symbol as c_Y then do nothing, otherwise:
   (d) Call $\mathtt{union}(c_X, c_Y)$.
   (e) Swap the roles of X and Y if necessary so that c_X is dead and c_Y is still alive.
   (f) Let $\mathcal{P}$ be the set of all productions involving X.
   (g) Remove all the productions in $\mathcal{P}$ from the grammar.

[3] A somewhat different $n \log n$ algorithm for congruence closure is described in [5].

(h) For each production $Z \rightarrow f(W_1, \cdots W_n)$ in $\mathcal{P}$ do the following.
   i. Let $c_{Z'}$ be `find`(c_Z) and for each W_i let $c_{W_i'}$ be `find`(c_{W_i})
   ii. If there is no production whose right hand side is $f(W_1', \cdots, W_n')$ then add the production $Z' \rightarrow f(W_1', \cdots, W_n')$.
   iii. If there is already a production $U \rightarrow f(W_1', \cdots, W_n')$ where U is different from Z' then add the equation $c_U = c_{Z'}$ to the queue S.

In the above procedure we assume that the find operation is such that `find`(c_X) is a canonical member of the equivalence class of c_X relative to the equivalence relation encoded in the union-find structure. In other words, `find`(c_X) is a constant c_Y such that `find`(c_Y) equals c_Y. We also assume that when two equivalence classes are merged with a union operation the canonical representative of the resulting equivalence class is selected to be one of the two previous canonical representatives.

The procedure maintains the invariant that no two productions in the grammar have the same right hand side, i.e., the grammar is always a congruence grammar. Furthermore, the equivalence relation encoded in the equations in the grammar, the queue, and the union-find procedure is maintained as a fixed invariant. To check this one must check that every added equation is derivable from previous equations and that every equation removed in step g is derivable from previous equations plus those equations added in step h. The procedure also maintains the invariant that every nonterminal in every production is "alive", i.e., if X appears in a production of the grammar then `find`(c_X) equals c_X. Every nontrivial execution of the main loop reduces the number of living nonterminal symbols, so the procedure must terminate. Furthermore, when the procedure terminates the equational theory encoded in the union-find structure can be dropped without altering the induced equivalence relation on external terms. To prove this consider an extended grammar that includes all productions of the form $X \rightarrow c_Y$ where X is a living nonterminal and `find`(c_Y) is c_X. The equation set of the extended grammar encodes the equivalence relation of the original grammar plus the equivalences in the union find structure. However, this extended grammar encodes the same intern function on external terms as the unextended grammar.

If we assume a bound on the number of arguments taken by function symbols, e.g., no function takes more than three arguments, and assume that hash table operations can be performed in unit time, then under an appropriate implementation of union-find it can be shown that the above procedure terminates in order $n \log n$ time where n is the number of productions in the original grammar. In practice the incorporation of a single new equation into a large grammar requires the manipulation of only a small subset of the grammar.

7 A Grammar Rewriting Algorithm

In this section we consider a fixed but arbitrary grammar rewriting system $<E, w>$, where w is a polynomial weight function, and give a procedure for computing all the ways in which a given grammar term can be rewritten under

$<E,\ w>$. The procedure is incremental so that if $<X,\ G> \mapsto_{<E,\ w>} <X',\ G'>$ then the set of possible ways of rewriting $<X',\ G'>$ can be computed incrementally from the set of possible ways of rewriting $<X,\ G>$. Incremental procedures can be defined by "inference rules" that are run in an incremental forward chaining manner. We first give rules for deriving the weight of nonterminal symbols as defined below.

> **Definition:** For each nonterminal Y of a congruence grammar the weight of Y, denoted $w[Y]$, is minimum weight of all terms generated by Y.

The following inference rule can be used to propagate bounds on weights.

$$
\begin{array}{l}
w[X_1] \leq w_i \\
\vdots \\
w[X_n] \leq w_n \\
Z \rightarrow f(X_1,\ \cdots,\ X_n) \\
\hline
w[Z] \leq P_f(w_1,\ \cdots,\ w_n)
\end{array}
$$

For constant symbols (functions of no arguments) the above inference rule can be used to generate a weight bound directly from a production of the form $Z \rightarrow c$. Weight bounds generated by constants can then be propagated to other symbols. One can show that, for any given nonterminal Y, the tightest bound that can be derived for $w[Y]$ using the above rule is in fact the weight of Y as defined above. In practice, simply running this rule over the entire grammar until the tightest bounds are derived seems to be an acceptable incremental algorithm for computing the weight of nonterminals. However, the theoretical worst case behavior of this this algorithm is quite bad. An $n \log n$ algorithm can be derived by placing derived bounds on a priority queue and processing the tightest bounds first.

Procedure to compute the weight of all nonterminals:

1. Initialize S to be the priority queue containing all pairs of the form $<Y, w>$ where the grammar contains the production $Y \rightarrow a$ where a is a constant and w is the weight of a.
2. Until S is empty, or until every nonterminal has been assigned a weight, do the following.
   (a) Remove a pair $<Y,\ w>$ from S such that w is the minimum weight of all pairs on S.
   (b) If Y has already been assigned a weight do nothing. Otherwise:
   (c) Assign Y the weight w.
   (d) For each production $Z \rightarrow f(W_1,\ \cdots,\ W_n)$ such that some W_i is Y and such that each W_i has been assigned a weight w_i, add the pair $<Z,\ P_f(w_1,\ \cdots,\ w_n)>$ to the queue S.

Each pair added to the queue has a larger weight than the last pair removed from the queue. This implies that the pairs removed from the queue have monotonically increasing weight. This, plus the assumed properties of the polynomial

weights, implies that for each nonterminal symbol Y the weight assigned to Y is the minimum over all the productions from Y of the weight computed from that production. One can check that by selecting productions that minimize weight that if Y has been assigned weight w then Y generates a term of weight w. Furthermore, one can prove by induction on the weight of a term s that the weight of s is at least as large as the weight assigned to $I[s]$, the symbol that generates s. Since the procedure only assigns a single weight to each nonterminal symbol the number of pairs placed on the priority queue is linear in the size of the grammar. Order n insertions and deletions from a priority queue can be done in $n \log n$ time. Assuming a bound on the number of arguments taken by any function symbol, the other operations in this procedure can be performed in order n time so the total running time is order $n \log n$.
The next step is to identify those nonterminals in the grammar that generate subterms of minimal representatives of $<X,\ G>$.

> **Definition:** A production $Z \rightarrow f(X_1,\ \cdots,\ X_n)$ is *minimal* if $w[Z] = P_f(w[X_1],\ \cdots,\ w[X_n])$.

> **Lemma:** If s is a minimal representative of the grammar term $<X,\ G>$ then s is generated by X in that subset of G which consists of just the minimal productions of G.

> **Definition:** We say that a nonterminal symbol Y is a *minimal subterm nonterminal* of a grammar term $<X,\ G>$ if either Y is X (X is a minimal subterm nonterminal) or G contains a minimal production $W \rightarrow f(Z_1,\ \cdots,\ Z_n)$ where W is a minimal subterm nonterminal and some Z_i is Y.

As an example consider the grammar term $<X,\ G>$ where G consists of the five productions

$$X \rightarrow f(Y),\ \ Y \rightarrow g(Z),\ \ Z \rightarrow a$$

$$Z \rightarrow h(W),\ \ W \rightarrow f(Z).$$

The production $Z \rightarrow h(W)$ is not minimal — it does not provide a smallest term generated by Z. However, all other productions are minimal, including the production $W \rightarrow f(Z)$ which provides a smallest term generated by W. However, the nonterminal W is not a minimal subterm nonterminal — it does not generate a subterm of a minimal representative of $<X,\ G>$.
The procedure for constructing matches uses a simple extension of the grammar.

> **Definition:** An *extended congruence grammar* is a congruence grammar such that, for each nonterminal X appearing in a production of the grammar, the grammar also contains the production $X \rightarrow c_X$.

We define a match to be a triple, `match`$[Y,\ s,\ \sigma]$, where Y is a nonterminal, s is a term, and σ is a substitution such that $I[\sigma[s]]$ is Y. Matches can be computed by starting with "basic matches" and creating new matches according to inference rules that generate new matches from old matches. For each minimal subterm nonterminal Y, and for each variable x occurring in E, we create the basic match $\mathtt{match}[Y,\ x,\ \{x \mapsto c_Y\}]$. To minimize the number of basic matches it is important to rename the variables in equations in E to minimize the total number of variables in E. Typically there will be no more than 3 or 4 variables in E. We also create basic matches for constant symbols that appear in E. For each constant a appearing in E we create the basic match $\mathtt{match}[I_G[a],\ a,\ \emptyset]$ where $\emptyset$ is the empty substitution. More complicated matches can be constructed using the following rule.

$$\frac{\begin{array}{l}\mathtt{match}[Y_1,\ s_1,\ \tau_1] \\ \vdots \\ \mathtt{match}[Y_n,\ s_n,\ \tau_n] \\ Z \rightarrow f(Y_1,\ \cdots,\ Y_n)\end{array}}{\mathtt{match}[Z,\ f(s_1,\ \cdots,\ s_n),\ \sigma]}$$

The substitutions are represented by finite lists of variable-value pairs. In all the substitutions constructed by the matching procedure the values assigned to variables are always internal constants. The above rule only applies when the substitutions τ_i agree on all shared variables — if τ_i and τ_j both contain a pair assigning a value to the variable x then they must both assign x to the same internal constant. The substitution σ is simply the union of the τ_i's. The above rule is also restricted to the case where Z is a minimal subterm nonterminal, the production $Z \rightarrow f(Y_1,\ \cdots,\ Y_n)$ is minimal, and $f(s_1,\ \cdots,\ s_n)$ is a term occurring in E.

The restrictions on the above inference rule are "nonmonotonic". The addition of a new production can cause other productions to go from being minimal to being nonminimal. When matches are computed incrementally as new productions are added it is possible that previously computed matches become "obsolete" in the sense that they were computed from productions that have now become nonminimal.[4] The failure to remove matches that are obsolete due to productions that are no longer minimal will cause minor overgeneration of matches. The time required to detect and remove these obsolete matches is probably greater than the time taken to perform any extra rewrites they cause. Rewrites generated by these obsolete matches are semantically sound and do no harm.

Finally, we construct "equate relations" of the form $c_Z \mapsto_E v$ where c_Z is an internal constant and v is a ground term involving internal constants. More specifically, if we derive $\mathtt{match}[Z,\ s,\ \sigma]$ and E contains either $s = t$ or $t = s$ where every free variable in t occurs in s, then we can derive $c_Z \mapsto_E \sigma[t]$. The

[4] Matches can also become obsolete if nonterminals or constant symbols involved in the match become "dead" due to generated equations between nonterminals. Matches that are obsolete due to references to dead nonterminals or dead constants are easily detected and eliminated.

set of derived equate relations of the form $c_z \mapsto_E v$ provide all the possible ways of rewriting the grammar. For each equate relation $c_Z \mapsto_E v$, the grammar can be rewritten by using the procedure of section 6 to equate c_Z and v.

8 Summary

Grammar rewriting is motivated by the desire to increase the success rate of attempts to prove equations in nonconfluent rewrite systems. Intuitively, the rewrite process generates a set of terms rather than an individual term, and by generating two sets of terms, rather than two classical normal forms, we increase the likelihood of proving the desired equation. Although grammar rewriting is primarily motivated by the need to handle nonconfluent theories, it also provides a new kind of canonical rewriting system. Under grammar rewriting there exist finite canonical systems for equational theories, such as idempotent semigroups, that do not have finite canonical systems under traditional notions of rewriting.

9 Acknowledgements

I would like to thank Robert Givan and Carl Witty for their help in the implementation of grammar rewriting in the Ontic theorem proving system. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

References

1. Franz Baader. Rewrite systems for varieties of semigroups. In *CADE-10, LNAI 449*, pages 381–395. Springer-Verlag, 1990.
2. L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In *Proc. Col. on Resolulution of Equations in Algebraic Structures*, 1987.
3. Robert S. Boyer and J Struther Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.
4. Leslie P. Chew. An improved algorithm for computing with equations. In *FOCS80*, pages 108–117. IEEE Computer Society Press, 1980.
5. Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpression problem. *JACM*, 27(4):758–771, October 1980.
6. J. Hsaing and M. Rusinowitch. On word problems in equational theories. In *ICALP-87, LNCS 267*, pages 54–71. Springer-Verlag, 1987.
7. J. P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15:1155–1194, 1986.
8. Dexter C. Kozen. Complexity of finitely presented algebras. In *Proceedings of the Ninth Annual ACM Symposium on the Theory of Compututation*, pages 164–177, 1977.
9. Ursula Martin and Tobias Nipkow. Ordered rewriting and confluence. In *CADE-10, LNAI 449*, pages 365–380. Springer-Verlag, 1990.

10. J. Mezei and J. B. Wright. Algebraic automata and context free sets. *Information and Control*, 11:3–29, 1965.
11. Greg Nelson and Derek Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–364, April 1980.
12. Gerald E. Peterson. Complete sets of reductions with constraints. In *CADE-10, LNAI 449*, pages 381–395. Springer-Verlag, 1990.
13. R. Shostak. An algorithm for reasoning about equality. *Comm. ACM.*, 21(2):583–585, July 1978.
14. W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics*, pages 133–164. MIT Press, 1990.