An Equality Theorem Prover Based on Grammar Rewriting

by

Serafim Batzoglou

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 1996

Author

Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by

David McAllester
Thesis Supervisor

Accepted by

F.R. Morgenthaler
Chairman, Department Committee on Graduate Students

An Equality Theorem Prover Based on Grammar Rewriting

by

Serafim Batzoglou

Submitted to the
Department of Electrical Engineering and Computer Science

May 28, 1996

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

We present the implementation of a term rewriting procedure based on congruence closure. The procedure can be used with arbitrary equational theories. It uses context free grammars to represent equivalence classes of terms. This representation is motivated by the need to handle equational theories where confluence cannot be achieved under traditional term rewriting. Context free grammars provide concise representation of arbitrary-sized equivalence classes of terms. In some equational theories where confluence cannot be achieved under term rewriting, it can be achieved under grammar rewriting, where the whole equivalence class of terms is ultimately represented with one context free grammar. Also, grammar rewriting can be used in any equational theory, besides the ones that have no confluent term rewriting system, to give an alternative form of rewriting. In the implementation of the procedure, we are concerned with the practical issue of how to build an efficient equality theorem prover using grammar rewriting. This involves representing the system in a clear and efficient way in the language C++. Also it involves constructing, and expreimenting with heuristics on the ordering of the queue of logical steps, or rewrites, performed by the system, in order to speed up and shorten the proofs without losing the desirable properties of confluence and/or termination.

Thesis Supervisor: Professor David McAllester
Artificial Intelligence Laboratory

Acknowledgments

I would like to thank my thesis advisor, David McAllester, for giving me the opportunity to work on theorem proving, and for his advice and support.

I would like to thank my family: my aunt Chrisoula, my mother Lambrini, and my sister Evi, for their love.

I dedicate this work to the memory of my father, Sylvestros.

Contents

Introduction

A rewriting system is an equality theorem prover, that takes as an input a set of equations, the axiom set, and two ground terms, i.e. two terms containing no free variables. Its job is to prove the two terms equal, or state that they are not equal under the equality set. Two important desirable properties of a rewriting system are confluence and termination. Confluence means the uniqueness of normal forms for equivalence classes of terms. In a confluent system two equal terms have the same normal form, so that in order to prove an equality between two terms, the rewriter just has to find the normal form of each term. Termination means that the rewriting is guaranteed to terminate. It follows easily that a confluent and terminating system is complete, i.e. decides all questions on the equality of two ground terms. Such a system is called canonical.

Most equational systems are nonconfluent with respect to traditional term rewriting, such as verifications done with the Boyer-Moore prover[2]. In these nonconfluent theories, the proof of true equations often fails. The initial motivation of Grammar Rewriting is to improve the success rate of rewriting in such theories. The idea behind this approach is that the success rate of proof attempts can be improved by rewriting a term to many equivalent ones in parallel. So, the canonical form of a term is the set of all equivalent terms, or more generally, the set of all equivalent terms that are minimal with respect to some weight function.

The problem with this idea is that equivalence classes of terms are very big in cardinality, sometimes infinite. Congruence grammars can serve as compact representations of such big sets. For example, under the equational theory that contains only commutativity, the equivalence class of a sum of n constants contains $O(2^n)$ terms, but a congruence grammar for generating it contains only O(n) productions. The rewriting algorithm described and implemented here is analogous to ordered rewriting systems [3], [4], [8], [12]. Grammar Rewriting is an attempt to associate terms with congruence grammars, and intuitively rewrite sets of terms, rather than individual terms. The hope is to improve the success rate of rewriting.

Another reason to pursue Grammar Rewriting is that under it there arises a new class of canonical rewrite systems, for theories that do not have canonical rewrite systems under traditional term rewriting. One example is the theory of idempotent semigroups, that is a canonical rewriting system under grammar rewriting, but does not have any finite canonical rewrite system under traditional term rewriting. In the full version of the paper [9], a canonical rewriting system for idempotent semigroups is described. It has been shown that there is no canonical term or word rewriting system for this theory [1].

The purpose of our research is to implement an equality theorem prover whose underlying term rewriting algorithm is based on grammar rewriting, in order to be able to do a practical study on the performance of this algorithm on various equational theories. In Chapter 1 we present this algorithm. Another purpose of this research is to explore ways of combining grammar rewriting with heuristic techniques that potentially improve the performance. In Chapter 2 we present some sample heuristics with which we experimented. We also present some rewriting systems on which we tested our theorem prover, and we finally include benchmark problems and statistics on the performance of the rewriter on these problems, under the heuristics that we constructed. In Appendix 1 we explain how we implemented the equality theorem prover, including the data structures that we used and the interface language.

The grammar rewriting procedure implemented here is somewhat similar to the rewriting technique described in [5], which is a congruence closure based rewriting procedure for nonground rewrite rules. That procedure, in contrast with the one implemented here, requires that the normal form of a term be a single term rather than an equivalence class represented by a congruence grammar.

The grammar rewriting procedure described here has also been incorporated into the Ontic verification system by David McAllester, Robert Givan, and Carl Witty. A main purpose of this paper though, is to perform isolated experimentation of the grammar rewriting procedure, including heuristics on the order of rewrites performed, as presented in Chapter 2.

Chapter 1

The Algorithm

1.1 Introduction

In this chapter we review the basic notions involving Grammar Rewriting, as described in [9].

The basic idea underlying the Grammar Rewriting algorithm is to use congruence grammars to represent equivalence classes of ground terms, under a finite set of ground equations. A congruence grammar is a context free grammar, where no two nonterminals lead to the same production. For example, the equivalence class of the constant symbol a under the equation set $\{a = f(a)\}$ is generated by the context free grammar:

$$\begin{aligned} X &\rightarrow f(X) \\ X &\rightarrow a \end{aligned}$$

Thus congruence grammars represent equivalence relations on terms. The relations that can be thus represented are the deductive closures of finite sets of ground equations.

Interning a ground term in a particular congruence grammar, involves finding the nonterminal that generates that term. Because we are working in congruence grammars, this nonterminal is guaranteed to be unique, if it exists. If there is no such nonterminal, then a fresh one is created. The procedure is recursive, so that in order to intern a term, its subterms are first interned.

A rewriting system is a set of equations, E, usually containing variables. Such a rewriting system attempts to show that two ground terms are equal, by showing that they both intern to the same

nonterminal. Starting with the empty grammar (or with any grammar), interning the ground terms s, t in order to prove $s = t$, results in a grammar G and two nonterminals, call them $I_G(s)$ and $I_G(t)$. If $I_G(s) = I_G(t)$ then $s = t$ under E. If not, then G is incrementally rewritten according to the equation set.

The rewriting is done as follows: A unification procedure is used to compute matches between nonterminals and patterns in the equation set. A pattern is a subterm of a term appearing in E, and a match is a triplet of a nonterminal X, a pattern p, and a substitution σ such that $\sigma(p)$ is a ground term and interns to X. At each point in time, all possible matches are computed. If there is a match $< X, p, s >$, $p = q$ is in E, and every free variable of q is sent to a nonterminal by σ, then $X = I_G(\sigma(p))$ and $I_G(\sigma(q))$ can be united, to become a single nonterminal. In that way, the equivalence classes of these two nonterminals collapse. At each point in time all possible rewrites of the grammar are held in a queue. There is an elaborate efficient algorithm for computing the new grammar G' that results from collapsing two nonterminals into one. The algorithm for computing the new grammar is based on the congruence closure algorithm described in [10], and runs in $O(nlogn)$ under the assumption that hash table operations take $O(1)$ time.

1.2 Congruence Grammars

In this section we define the basic concepts that we use in the grammar rewriting procedure, namely finite congruence grammars and finite ground sets. It turns out that these two concepts are interchangeable, meaning that one can go from a finite congruence grammar to a finite ground equality set, and conversely. A procedure is given for incrementally building a congruence grammar that models a ground equality set.

- A **symbol** is defined to be either:
  A **variable**, denoted by x_i where i ranges over the natural numbers, or
  A **function symbol** of arity[1] n, where n ranges over the natural numbers. We assume an infinite set of function symbols of each arity.

- A **term** t is defined recursively, to be:
  (a) A variable, or
  (b) A function application $f(t_1, ..., t_n)$ where n ranges over the natural numbers, t_i for $0 \leq i \leq n$

[1] Arity is the number of arguments

is a term, and f is a function symbol of arity n. In case $n = 0$, $t = f()$ is a **constant**.
A **ground term** is defined recursively, to be a function application $f(t_1, ..., t_n)$ where n ranges over the natural numbers and t_i for $0 \leq i \leq n$ is a ground term. A **subterm** of a term $t = f(t_1, ..., t_n)$ is either t, or a subterm of some t_i for $1 \leq i \leq n$. A **proper subterm** of t is a subterm of some t_i. An **immediate subterm** of t is one of the terms t_i.

- A **production** p is defined to be an object of the form $X \rightarrow f(Y_1, ..., Y_n)$ where X and Y_i are nonterminals (we assume an infinite collection of nonterminals), n ranges over the natural numbers, and f is a function symbol of arity n. If $n = 0$, then p is a **ground production**. The **right hand side (r.h.s.)** of p is $f(Y_1, ..., Y_n)$. We define $from(p)$ to be the nonterminal X, and $To(p)$ to be the set $\{Y_1, ..., Y_n\}$ of nonterminals that appear in the r.h.s. of p.

- A **congruence grammar** G is finite a set of productions, where no two productions have the same right hand side.

- A nonterminal X **produces** (or **generates**) a ground term t in a congruence grammar G, if $t = f(t_1, ..., t_n)$, and:
(a) $n = 0$ and there is a ground production $p = X \rightarrow f()$ in G, or
(b) $n > 0$, and there is a production $p = X \rightarrow f(Y_1, ..., Y_n)$ in G such that for each $1 \leq i \leq n$, Y_i produces t_i.
In both cases (a),(b), the production in question is said to generate t. The grammar G is said to generate t, if some production in G generates t.

Lemma: A ground term t is generated by at most one nonterminal of a congruence grammar G.
Proof: Case 1: If t is a constant, then it is generated by at most one nonterminal, else there are two productions in G with the same right hand side, namely t.
Case 2: If t is $f(t_1, ..., t_n)$, $n > 0$, and $X \rightarrow f(Y_1, ..., Y_n)$, $Z \rightarrow f(W_1, ..., W_n)$ are two productions generating t, then Y_i and W_i both generate t_i for $1 \leq i \leq n$, so by inductive assumption $Y_i = W_i$. But this is a contradiction, because then both productions have the same right hand side. Δ

- We define the **equivalence relation** with respect to a congruence grammar G, on terms generated in G: two terms s, and t generated in G are equivalent in G, if $I_G(s) = I_G(t)$ where I_G is the function on terms generated in G that returns the nonterminal that produces them. We extend the function I_G to ground terms not generated in G. In order to do that, we will describe a procedure that extends G to a grammar G' that generates t, and $I_G(t)$ is defined

to be $I_{G'}(t)$. We will call this procedure, applied on t, "*interning*" t [2]:

Interning $t = f(s_1, ..., s_n)$ **in** G:

1. Let $Y_1, ..., Y_n$ be the result of recursively interning $s_1, ..., s_n$, if $n > 0$.
2. If there is a production $X \rightarrow f(Y_1, ..., Y_n)$ in G, return $I_G(t) = X$.
3. If not, select a nonterminal X not appearing in G, and create the production $X \rightarrow f(Y_1, ..., Y_n)$, adding it to the grammar to get G'. Return $I_G(t) = X$.

Finally we define the **equivalence relation** with respect to G, on any ground term, to be the one defined by the partition function I_G on ground terms.

Congruence grammars are capable to express any equivalence class defined by a finite set of ground equations. This is expressed in the following result:

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

The proof of this theorem is based on well known algorithms for congruence closure[10]. Conversely, we define $Eq(G)$ to be the set of equations that models the equivalence relation on terms expressed by the congruence grammar G. For each nonterminal X appearing in the grammar G, construct a new constant symbol c_X. An **extended congruence grammar** G is one that, for each nonteminal X appearing in G, contains a production $X \rightarrow c_X$ where c_X is a special constant for nonterminal X. Then if $Eq(G)$ is the set of equations of the form $c_X = f(c_{Y_1}, ..., c_{Y_n})$ where G contains the production $X \rightarrow f(Y_1, ..., Y_n)$, we get the following result:

[2] At this point we abuse the notation, because the procedure we are going to describe is nondeterministic in terms of exactly which nonterminal is returned, and as a consequence I_G is no longer a function. But the nondeterminism on the interning procedure can be raised easily, and in any case the equivalence relation defined is not altered.

Lemma: For any congruence grammar G, and any two ground terms s and t, $I_G(s) = I_G(t)$ if and only if $Eq(G) \models s = t$.

Proof of Lemma: By induction on the size of a term s, $Eq(G) \models s = c_{I_G(s)}$. This is clear for constants, because $a = c_{I_G(a)} \in E$ for every constant produced by G. Now let $X \to f(Y_1, ..., Y_n)$ be a production producing $s = f(t_1, ..., t_n)$. Then by inductive assumption, for $1 \leq i \leq n$, $Eq(G) \models t_i = c_{Y_i}$. Also, by definition $c_X = f(c_{Y_1}, ..., c_{Y_n}) \in Eq(G)$. So $Eq(G) \models s = c_x$. To show the converse, extend G by all productions of the form $X \to c_X$ to get the extended congruence grammar G'. The intern function $I_{G'}$ is a partition function that defines an equivalence relation on terms generated by G'. This equivalence relation provides a semantic model for $Eq(G')$, and agrees with the equivalence relation defined by I_G on the terms generated by G, because the two functions agree on these terms. So if $I_G(s) \neq I_G(t)$ then $Eq(G) \not\models s = t$.Δ

Below is a procedure that incorporates ground equations into a congruence grammar. This procedure is also presented in [9] and is very similar to a congruence closure procedure presented in [10]. The procedure starts with an extended congruence grammar G, and an equation $s = t$ where s and t are ground terms. It computes $Gr(Eq(G) \cup \{s = t\})$. Before describing this procedure we need to define a union-find structure on nonterminals, whereby there is a function *find* from nonterminals to nonterminals and calling *union* on two nonterminals, alters the outputs of *find*. We further assume that $find(X)$ appears in the grammar, and $find(find(X)) = find(X)$ for any nonterminal X appearing in the grammar.

Calling union **on the nonterminals** X **and** Y, $union(X, Y)$:

1. Call $choose(X, Y)$, that returns either X, or Y[3].. Say $Z = choose(X, Y)$, and W is the other one of X, Y.

2. If $find(Z) = Z$, set $find(Z) = W$ and do nothing else.

[3]This function, *choose*, returns the nonterminal that is going to be killed. In order for the algorithm to run provably in $O(nlogn)$ time it is important to implement this function so as to always choose the "smallest" nonterminal, under some particular size function. We do not get into this in detail here.

3. If $find(Z) = V \neq Z$, call $union(V, W)$, and then set $find(Z)$ to be $find(V)$.

- We say that a nonterminal X is **dead** if $find(X) \neq X$. Else, we say that X is **alive**.

Computing $Gr(Eq(G) \cup \{s = t\})$**:**

1. Intern s and t, and let Z and W be the nonterminals $I_G(s)$ and $I_G(t)$.
2. Initialize Q to be a queue of ground equations, containing the single equation $c_Z = c_W$.
3. While Q is not empty, do:
   (a) Remove an equation $c_Z = c_W$ from Q.
   (b) Intern c_Z, c_W to obtain Z and W.
   (c) Let $X = find(Z)$ and $Y = find(W)$.
   (d) If $X = Y$ do nothing, otherwise:
   (e) Call $union(X, Y)$.
   (f) Let P be the set of all productions involving X[4].
   (g) Remove all productions in P from the grammar.
   (h) For each production $Z \to f(W_1, ..., W_n)$ in P do:
      i. Let Z' be $find(Z)$ and for each W_i, $1 \leq i \leq n$ let W_i' be $find(c_{W_i})$.
      ii. If there is no production whose right hand side is $f(W_1', ..., W_n')$, add the production $Z' \to f(W_1', ..., W_n')$ to the grammar.
      iii. If there is already a production $U \to f(W_1', ..., W_n')$ in the grammar, where U is different from Z', then add the equation $c_U = c_{Z'}$ to the queue Q.

1.3 Grammar Rewriting

- A **grammar rewriting system** is a finite set of term equations E, where each element of E is an equation between two terms (that usually contain variables).
- A **ground substitution** σ on a congruence grammar G is a function from a subset of the set of variable symbols, to the set of ground terms that are generated by G.

[4] X is now dead.

- On a grammar rewriting system E, and extended congruence grammar G such that every ground term appearing in E is produced in G, a **match** is a triplet, $match[Y, s, \sigma]$ where Y is a nonterminal, s is a term, and σ is a substitution such that $Y = I_G(\sigma(s))$. Moreover, for any variable x in the domain of σ, $\sigma(x)$ equals the special constant c_Z for some nonterminal Z appearing in G. Matches can be computed by starting with "basic" matches, and creating new matches according to an inference rule that generates new matches from old matches. We define the **match set** $M(G, E)$ of G and E, to be the maximal set of matches constructed as follows:

  1. For each nonterminal X appearing in G and for each variable x occurring in E, we create the basic match $match[X, x, x \rightarrow c_X]$.
  2. For each constant a appearing in E we create the basic match $match[I_G(a), a, \{\}]$.
  3. We apply the rule

$$\frac{\begin{array}{c} match[Y_1, s_1, \sigma_1] \\ \vdots \\ match[Y_n, s_n, \sigma_n] \\ X \rightarrow f(Y_1, ..., Y_n) \end{array}}{match[X, f(s_1, ..., s_n), \sigma]}$$

  where for any two substitutions σ_i, σ_j with $1 \leq i, j \leq n$ we require that if $\sigma_i(x)$ and $\sigma_j(x)$ are both defined, they are the same constant. We also require that the term $f(s_1, ..., s_n)$ is a term occurring in E. Then, we define $\sigma = \bigcup_{i=1}^{n} \sigma_i$[5].

  We say that Y matches s under the substitution σ, if $match[Y, s, \sigma]$ is in $M(G, E)$.

Lemma: $M(G, E)$ is finite.

Proof: The term in a match triplet is a subterm appearing in E. There are finitely many of them, and we can prove that $M(G, E)$ is finite by recursing on the structure such terms. Clearly there are finitely many matches involving constants and variables. The rest of the matches are obtained by the one propagation rule given above, and assuming that the set of the possible matches involving

[5] We are abusing the notation, because σ is a function and not a set, but it should be clear what we mean.

each s_i in the premises of the rule is finite, it follows that the set of possible matches to the term $f(s_1, ..., s_n)$ is finite.Δ

- A **rewrite** r on G, E, and $M(G,E)$, an extended congruence grammar, grammar rewriting system, and the associated match set, is a triplet $< X, s \rightarrow t, \sigma >$ where X matches s under σ, and either:
  (a) $s = t \in E$, and every free variable of s appears in t, or
  (b) $t = s \in E$, and every free variable of t appears in s.
  A **nontrivial rewrite** r on G, E, and $M(G,E)$ is a rewrite $< X, s \rightarrow t, \sigma >$ such that $I_G(\sigma(t)) \neq X$.

Notice that this definition of rewrites implicitly defines an ordering in each of the equations in E. That is to say, we don not allow rewrites to a term containing a variable x, from a term not containing the variable x. In our implementation of rewriting we did something different: We only allowed rewrites that satisfy condition (a). The above definition of rewrites can be translated to the one we implemented, as follows: For every equation $s = t \in E$, put $t = s$ in E as well. In our treatment, we can have a direction on the equations, so that basically we are dealing with rewrite rules. Examples where this matters will be given in Chapter 2. In this chapter, we assume that equation sets are treated by the rewriting system as explained above, i.e. that both directions of an equation are allowed by default, except when the restriciton on the free variables prohibits one of the two directions[6].

Lemma: The set of rewrites on G, E, and $M(G,E)$ is finite.
Proof: $M(G,E)$ is finite, and t ranges over a finite set of terms.Δ

- $Q(G,E)$, where G is a congruence grammar, E is a rewriting system, and $M(G,E)$ is the associated match set, is a **rewrite queue** that contains every nontrivial rewrite r on G, E, and $M(G,E)$. $Q(E,G)$ contains a function $index_{Q(G,E)}$, that assigns to each rewrite r an integer $index_{Q(G,E)}(r)$ which is the number of rewrites that were inserted before r.

- A deterministic **grammar rewriter**, GRR, is a module that behaves as follows:

  1. It takes as input a quadruple $< G, E, M(G,E), Q(G,E) >$ where G is a congruence grammar, E is a rewrite system, and $M(G,E)$, $Q(G,E)$ the associated match set and

[6] Or no direction is allowed in the equation $s = t$, if both s and t contain variables that the other term does not.

rewrite queue. It also takes as an input a special ground equation $s = t$, called *goal*, such that s and t are generated in G.

2. It contains a function $extract_{GRR}$ that takes as inputs the inputs of GRR, and extracts an element $r =< X, s, t, \sigma >$ from $Q(G, E)$.
3. It contains a procedure *incorporate* that takes as inputs r and the inputs of GRR and behaves as follows:
   (a) It incorporates the ground equation $c_X = c_{I_G(\sigma(t))}$ into G, to get the new congruence grammar $G' = Gr(Eq(G) \cup \{c_X = c_{I_G(sigma(t))}\})$.
   (b) It updates $M(G, E)$ to obtain $M(G', E)$.
   (c) It updates $Q(G, E)$ to obtain $Q(G', E)$, so that $index_{Q(G',E)}(r) = index_{Q(G,E)}(r)$ for any rewrite $r =< X, s, t, \sigma >$ such that the match $match[X, s, \sigma]$ is in the old match set $M(G, E)$, and consequently r is in the old queue $Q(G, E)$[7].

Notation: Call the result of this one step rewriting $< G', M(G', E), Q(G', E) >= GRR(< G, E, M(G, E), Q(G, E) >, goal)$.

Such a grammar rewriter is very general, as to the order in which rewrites are performed. We insist that extraction is deterministic, but virtually any other strategy can be implemented under the definition above. For instance, treating differently each equation (different priorities) can be incorporated, by hardwiring information on GRR.

- A **grammar term** is a structure $< X, G, Q(G, E), E >$ where G is a congruence grammar, X is a nonterminal that appears in G, E is a rewriting system, and $Q(E, G)$ is the associated rewrite queue.
- We define the one step rewrite relation $\rightarrow_{GRR,goal}$ on grammar terms so that

$$< X, G, Q(G, E), E > \rightarrow_{GRR,goal} < X', G', Q(G', E), E >$$

provided that

$$< G', M(G', E), Q(G', E) >= GRR(< G, E, M(G, E), Q(G, E) >, goal)$$

[7]Here we insist that the *index* of the rewrites is preserved, because in the next call of the rewriter we may need the information of which rewrite was inserted first in the queue. This is important for strategies of extracting rewrites from the queue, that use information on the order with which the rewrites were inserted. Example: FIFO queue.

where $extract_{GRR}(X, G, Q, E, goal)$ is a rewrite $r =< X, u, v, \sigma >$, and $X' = I_{G'}(\sigma(u))$, and *goal* is a ground equation $s = t$ such that s and t are generated by G.

- We say that a grammar term $< X, G, Q(G, E), E >$ is in **normal form** if $Q(E, G)$ is empty. If the context makes clear what E is, we can alternatively denote such a term as $< X, G >$.

Lemma: Let $< X, G, Q(G, E), E >$ be a grammar term, where $Q(G, E)$ contains no nontrivial rewrites. Let $\rightarrow_{GRR,goal}$ be a one step rewrite relation such that *goal* is a ground equation whose ground terms are generated by G. Say $Q(G, E)$ contains k elements. Then after $\leq k$ successive applications of $\rightarrow_{GRR,goal}$ the term $< X, G, Q'(G, E), E >$ is reached, which is in normal form.

Proof: Observe that only nontrivial rewrites modify the grammar G, or the nonterminal X.Δ

As an example of rewriting, let C be the equation set consisting of the commutative law $x+y = y+x$ where x, y are variables. Let G be the grammar consisting of the productions $X_1 \rightarrow a_1 + X_2$, ..., $X_{n-1} \rightarrow a_{n-1} + X_n$, $X_n \rightarrow a_n$, i.e. the grammar generated by interning the term $a_1 + (a_2 + (... + a_n))$ starting with the empty grammar. In the collection of all grammar rewriters, we have that the grammar term $< X_1, G, Q(G, E), C >$ rewrites to the normal form $< X_1, G', Q(G', E), C >$ where G' is the grammar consisting of the productions of the form

$$\begin{aligned} X_i &\rightarrow a_i + X_{i+1} \\ X_i &\rightarrow X_{i+1} + a_i \end{aligned}$$

together with the production

$$X_n \rightarrow a_n$$

and *goal* is any pair of terms produced by nonterminals among $\{X_i \text{ where } 1 \leq i \leq n\}$. The reason this is a normal form, is that $Q(G', E)$ is empty: there are no possible nontrivial rewrites.

- A grammar rewriting system E is called **terminating** with respect to a grammar rewriter collection S, if there are no infinite rewrite chains of the form

$$< X_1, G_1, Q(G_1, E), E > \rightarrow_{GRR,goal} < X_2, G_2, Q(G_2, E), E > \rightarrow_{GRR,goal} \cdots$$

  where $GRR \in S$, G_1 is any congruence grammar, X_1 appears in G_1, and *goal* is any goal whose terms are produced in G_1.

Certain restrictions on S ensure that rewriting always terminates. In [9] a weight function is used which assigns weights to all terms produced by a grammar, and assigns weights to each nonterminal according to the minimal term produced by that nonterminal. Then, rewriting is allowed only for certain terms that satisfy a minimality condition. Assuming that the weight function is polynomial on the size of the term, rewriting is guaranteed to terminate. Here we are not going to get into the details of this particular machinery.

- A grammar rewriting system E is called **complete** with respect to a grammar rewriter collection S, if, assuming that $Eq(G)$ contains no ground equations unprovable by E, then for any pair of terms s, t, and for any normal forms $< X, G >$, and $< Y, G >$ with respect to E, of s and t respectively, we have that $s = t$ under E if and only if $X = Y$.

- A grammar rewriting system is called **canonical** with respect to a grammar rewriter collection S if it is terminating and complete with respect to S.

A grammar rewriter belonging to a collection S, such that E is canonical with respect to S, provides a decision procedure for the equation theory E: Let $s = t$ be a goal. Intern s in the empty grammar, to obtain G_s. Intern t in G_s to obtain $G_{s,t}$. Let $< X, G, Q(G, E), E >$ be a normal form of $< I_{G_s,t}(s), G_{s,t}, Q(G_{s,t}, E), E >$ with respect to $GRR \in S$ and *goal* be $(s = t)$. Let $< Y, G', Q(G', E), E >$ be a normal form of $< I_G(t), G, Q(G, E), E >$ with respect to $GRR, goal$. Then, $< I_{G'}(s), G', Q(G', E), E >$ is a normal form of s with respect to $GRR, goal$. Then $I_{G'}(s) = I_{G'}(t)$ iff $s = t$ under E.Δ

The above paragraph is indicative of the way we perform grammar rewriting in our system. To prove a goal $s = t$, we intern s in the empty grammar, then we intern t in the resulting grammar, and then we perform grammar rewriting, until (1) s and t intern to the same nonterminal, in which

case we are done, or (2) both s and t have reached their normal forms[8], in which case the proof attempt failed, or (3) we stop the procedure under a certain limit in the number of steps, or in the time it takes, in which case we do not know the result.

[8]We are abusing the terminology here, but it is obvious what we mean

Chapter 2

Queue Ordering Heuristics

The order in which the rewrites are performed is important in Grammar Rewriting. It affects the performance, and even the termination/confluence properties of the rewrite system. We give an example below, where two different ordering strategies perform very differently. In particular, one of the strategies terminates, while the other diverges.

To keep the example simple, we choose the following theory, and its associated rewriting system: Consider the subset of group theory that proves two terms s and t to be equal, using only the associative law, and the *compressing* identity and inverse laws. By *compressing*, we mean that these laws can only be performed in the "compressing" direction, i.e. $*(id, x) \rightarrow x$, $*(x, inv(x)) \rightarrow id$, etc.

Notational convention A rewrite rule denoted by $s = t$ is treated as an equation, meaning that both directions $s \rightarrow t$ and $t \rightarrow s$ are valid, always under the restriction given in the definition of a rewrite, in Chapter 1. A rewrite rule denoted by $s \Rightarrow t$ is oriented, meaning that the direction $t \rightarrow s$ is not allowed.

The following is the corresponding rewrite system:

Compressing part of group theory:

$$\begin{aligned} *(*(?x, ?y), ?z) &= *(?x, *(?y, ?z)) \\ (id, ?x) &\Rightarrow ?x \\ (?x, id) &\Rightarrow ?x \\ (inv(?x), ?x) &= id \\ (?x, inv(?x)) &= id \end{aligned}$$

It is easy to see that a $FIFO$ ordering of the queue of rewrites will prove any statement in this theory. The reason is that with a $FIFO$ strategy any possible rewrite on the terms s, t, and their subterms will be eventually performed. With a little care of where to stop a proof if it does not seem to succeed, it is simple to make this into a terminating, hence complete, rewriting system.

Consider now a $LIFO$ ordering. Say the problem instance is:

$$\begin{aligned} Let \quad t &= *(inv(b), *(*(*(inv(a), a), b), c)) \\ Let \quad s &= c \\ Show: \quad t &= s \end{aligned}$$

In order for this to succeed, the following rewrites need to take place:

1. $*(inv(a), a) \rightarrow id$
2. $*(id, b) \rightarrow b$
3. $*(inv(b), *(b, c)) \rightarrow *(*(inv(b), b), c)$
4. $*(inv(b), b) \rightarrow id$
5. $*(id, c) \rightarrow c$

Say (1) is performed first, and at any point later, (2) is performed before any associativity involving $inv(b)$. Then, the nonterminal $X = I_G[*(*(*(inv(a), a), b), c)]$ produces also the term $*(*(*(inv(a), a), *(*(inv(a), a), b)), c)$. Since the queue is $LIFO$, the associativities that are going to "bring together" $*(a, inv(a))$ in this last term, are going to be performed before (3). Then, $*(a, inv(a)) \rightarrow id$ will make id equivalent to both $*(a, inv(a))$ and $*(inv(a), a)$. But then, $X = I_G[*(*(id, *(id, b)), c)]$, and before any associativity involving $inv(b)$ is performed, it will be shown that $X = I_G[*(*(*(id, id), b), c)]$, and then because $*(id, id) = *(*(inv(a), a), *(a, inv(a)))$, before (3) is performed associativities in this last term will be performed, and will cause the term $*(a, a)$ to be interned. Now it is not hard to see that the $LIFO$ rewriting of terms emerging from the nonterminal $I_G[*(*(id, *(id, b)), c)]$ will eventually cause the interning of all the terms of the form a^n, $n > 0$, and this is an infinite number of terms. All this is going to take place before (3) has a chance to be performed. So, this rewriting system is not terminating with $LIFO$ heuristics.

We define the property of fairness for a strategy of extracting rewrites from the queue, that is going to help us analyze the termination properties of various weight heuristics. In order to do that, we define a clear execution sequence, for a strategy:

- A **clear execution sequence** for a strategy, starts with interning a goal $s = t$ in the empty grammar, and then performing rewrites $r_1, r_2, ...$ which are extracted from the queues $Q_1(G_1, E), Q_2(G_2, E), ...$ according to the strategy. That is, no term is interned at any point, besides the goal and the terms interned by the algorithm.
- An extraction strategy satisfies **fairness** if for every clear execution sequence which does not terminate, for every i, and every rewrite r in $Q_i(G_i, E)$, there is a $j \geq i$ such that r is extracted and performed at step j.

Informally, in the definition of a clear execution sequence, we ensure that there are never inserted in the queue any rewrites that could possibly deteriorate the performance of the heuristics in question. That is to say, there are no rewrites from terms that cannot be obtained by interning only the terms in the goal, and no other term. Notice that $FIFO$ satisfies fairness, the integer j for r in $Q(G, E)$ being the number of rewrites inserted before r. $LIFO$ does not satisfy fairness, and a counterexample is given above.

2.1 Definitions of Weights

- Define the **size** of a term $t = f(s_1, ..., s_n)$, $size(t)$, as follows:
  if $n = 0$ **then** $size(t) = 1$;
  else $size(t) = 1 + \sum_{i=1}^{n} size(s_i)$;
  Notice that this is equal to the number of function symbols in t.

- Define the **size** of a pattern p, $size(p)$, to be likewise the number of function symbols in p.

- Define the **size**, $size(X)$, function on nonterminals as follows: it is the pointwise maximum value that satisfies the following constraints: (to avoid confusion, s, t are always terms below, while X, Y are always nonterminals).

Size Function on Nonterminals, pointwise max $size(X)$ **s.t.:**

$$\frac{size(t) = m \qquad X \rightarrow t}{size(X) \leq m}$$

$$\frac{\begin{array}{c} size(Y_1) \leq i_1 \\ \vdots \\ size(Y_n) \leq i_n \\ X \rightarrow f(Y_1, ..., Y_n) \end{array}}{size(X) \leq 1 + \Sigma_{j=1}^{n} i_j}$$

We define a **minimal production** of a nonterminal X to be a production $p = X \rightarrow f(Y_1, ..., Y_n)$ such that $size(X) = 1 + \Sigma_{j=1}^{n} size(Y_j)$. By a simple induction one can show that there is such a production, for every X.

- Define the **context size**, $cosz(X, R)$, function on nonterminals as follows (both X and R are nonterminals): it is the pointwise maximum value satisfying the following constraints:

Context Size Function on Nonterminals, pointwise max $cosz(X)$ s.t.:

$$cosz(R, R) = 0$$

$$\begin{gathered} cosz(X, R) \leq m \\ size(Y_1) \leq i_1 \\ \vdots \\ size(Y_n) \leq i_n \\ \hline X \rightarrow f(Y_1, ..., Y_k, ..., Y_n) \\ cosz(Y_k, R) \leq m + \Sigma_{\substack{j=1 \\ j \neq k}}^{n} i_j \end{gathered}$$

Notice that $cosz(X, Y)$ is infinity, if there is no *path* from Y to X, where a *path* from a nonterminal Y to a nonterminal X is a sequence of productions $p_1, ..., p_n$ such that (1) p_1 comes from Y, (2) for each $1 < i \leq n$ p_i comes from a nonterminal appearing in the right-hand side of p_{i-1}, and X appears in the right-hand side of p_n. Notice that $cosz(X, Y)$ is the size of the smallest term $C[R]$ containing the term R such that $Y = I_G(R)$ and $X = I_G(C[R])$, minus the size of R.

- Define the **size** of a substitution σ, $size(\sigma)$ to be: (Let $nvars$ to be the number of variables in the grammar in question, and $var(j)$ for $1 < j < nvars$ to be the jth variable of the grammar).

$$size(\sigma) = \Sigma_{j=1}^{nvars} size(\sigma(var(j)))$$

Now we are ready to define some weight functions on rewrites, that can serve to order the queue of rewrites with the elements with smallest weight having highest priority. Let the rewrite be $r =< X, L \rightarrow R, \sigma >$. The first such weight function involves the size of the term $\sigma(R)$ to which the nonterminal X rewrites, and is therefore:

$$Weight_{size}(r) = size(\sigma(R))$$

The second weight function involves the context size of X with respect to the *goal*. Specifically, let the goal be to show that $t = s$. Let $Y = I_G(t)$ and $Z = I_G(s)$. Then, we define $Weight_{cosz}(r)$ to be:

$$Weight_{cosz}(r) = min\{cosz(X,Y), cosz(X,Z)\}$$

The third weight function is more complicated. We want to express how "far" a rewrite is from using the minimal productions of the grammar. Accordingly, let $r =< X, L \rightarrow R, \sigma >$, let the goal be $t = s$, and let $Y = I_G(t)$ and $Z = I_G(s)$ we define $Weight_{distance}(r)$. Again, this function can be infinity for some nonterminals:

$$Weight_{distance}(r) = min\{cosz(X,Y) - size(Y), cosz(X,Z) - size(Z)\} + size(\sigma(L))$$

Notice that for any rewrite $r =< X, L \rightarrow R, \sigma >$ and for any $Y = I_G(t)$ where either $s = t$ or $t = s$ is the goal, the quantity $cosz(X,Y) - size(Y) + size(\sigma(L))$ is equal to 0, if we are using only minimal productions to get a path to the term $\sigma(L)$ from the nonterminal Y. This observation justifies the above definition of the $Weight_{distance}$ function.

2.2 Queue Orderings

The following are the different queue orderings we used:

- **FIFO:** According to this ordering, the rewrites are extracted from the queue in the same order in which they are inserted. This ordering satisfies fairness, as we already proved.
- **LIFO:** According to this ordering, the rewrites are extracted from the queue in the opposite order in which they are inserted, last in first out. We already demonstrated that this ordering does not satisfy fairness.
- **SIZE:** According to this ordering, the rewrites are extracted from the queue with priority given to the ones that are minimal with respect to the $Weight_{size}$ function. In case of a tie, priority is given to the ones that are minimal with respect to the $Weight_{distance}$ function. In case of a further tie, priority is given to the one inserted first.

Claim: The $SIZE$ ordering satisfies fairness.

Proof: (Informal) First, for any k, notice that there is a bound on the number of steps that can be performed without generating a nonterminal X such that $size(X) > k$. This is true because the nontrivial unions that can be performed between nonterminals whose minimal representatives are of size bounded by k, is bounded by the number of different ground equalities between terms of size at most k. Second, notice that when interning the two sides $\sigma(L)$ and $\sigma(R)$ of a rewrite $r =< W, L \rightarrow R, \sigma >$, only the interning of $\sigma(R)$ produces any new nonterminals, because the interning of $\sigma(L)$ has already been performed, and returned W. Then, consider $size(r)$. This is equal to $size(\sigma(R))$, thus any new nonterminal generated by computing $Gr(Eq(G) \cup \{\sigma(L) = \sigma(R)\})$ has size bounded by $size(r)$. Finally, for any nonterminal r in the queue, we conclude that there can be performed only a finite number of nontrivial unions before r is small enough to be extracted from the queue.Δ

- **DISTANCE:** According to this ordering, the rewrites are extracted from the queue with priority given to the ones that are minimal with respect to the $Weight_{distance}$ function. In case of a tie, priority is given to the ones that are minimal with respect to the $Weight_{size}$ function. In case of a further tie, priority is given to the one inserted first.

Claim: The $DISTANCE$ ordering satisfies fairness.

Proof: (Informal) Say that $s = t$ is the goal, and $X = I_G(s)$, $Y = I_G(t)$. Let r be the rewrite $< W, L \rightarrow R, \sigma >$. Then notice that $Weight_{distance}(r) = n$ is finite, because it is easy to see that rewriting the goal always produces terms that are a finite distance from the goal. Moreover, for any rewrite in the queue, $Weight_{distance}(r) \geq (size(\sigma(L)) - size(X))$. Finally, notice that there are only a finite number of terms in the language, such that $size(t) \leq n$. The number of such terms can be computed from the language of terms alone. If all rewrites are performed starting with such terms $t = \sigma'(L')$, then $DISTANCE$ has to choose to perform r.Δ

- **CONTEXT:** According to this ordering, the rewrites are extracted from the queue with priority given to the ones that are minimal with respect to the $Weight_{cosz}$ function. In case of a tie, priority is given to the ones that are minimal with respect to the $Weight_{distance}$ function. In case of a further tie, priority is given to the one inserted first.

Steps at most:	5	10	50	100	256
FIFO	1	3	9	14	17
LIFO	0	0	2	2	2
SIZE	1	4	11	16	16
CONTEXT	0	1	3	3	5
DISTANCE	1	5	9	12	15

Table 2.1: Comparison of five heuristic strategies, in 17 benchmark problems

CONTEXT heuristics does not satisfy fairness. To obtain an example that breaks the condition, consider a theory that has the axiom $x = x^3$, and a case where $t = a$ where a is a constant, is the goal. Initially the large term t is interned, producing some rewrites of considerable context weight. Then, a is interned, producing the rewrite $a \rightarrow a^3$, of small weight. Finally, if a term s has weight n, and $s \rightarrow s^3$ is performed, then s^3 has weight n as well.

For many rewrite systems, the *FIFO*, *SIZE*, or *DISTANCE* queue orderings guarantee fairness while some other orderings do not. An easy fix in such cases, if one still wants to try the other orderings, is to combine them with the *FIFO*, ordering. That is, every certain constant number of steps, a rewrite is performed according to the *FIFO*, ordering. But from the experimental analysis, we concluded that good performance, in almost all cases, assumes fairness of the strategy. This is demonstrated in Table 2.1, that summarizes the performance of the different strategies in 17 benchmark problems. In this table, the different columns correspond to the different upper bounds on the number of nontrivial rewrite steps, as given in the first row. In the rest of the chapter we will elaborate on the experimentation with the rewriter on these benchmarks.

2.2.1 Other types of queue orderings

There are other types of queue orderings apart from taking into account only the order that rewrites are inserted into the queue, and the size/context of the terms expressed. A class of orderings that is promising is the orderings that give different priorities to different rewrite rules. Such orderings are possible according to the definition of a grammar rewriter we gave in Chapter 1. Another possibility is combining some priority on the rewrite rules, with the above weight functions. According to that, each rewrite rule would have a "weighted priority" that could be additive, or multiplicative, to the above weight functions. There are numerous possibilities, and most probably some of them would

$$
\begin{array}{rcl}
((?x, ?y), ?z) & = & *(?x, *(?y, ?z)) \\
(id, ?x) & \Rightarrow & ?x \\
(?x, id) & \Rightarrow & ?x \\
(inv(?x), ?x) & = & id \\
(?x, inv(?x)) & = & id \\
?x & \Rightarrow & *(*(inv(?x), ?x), ?x)
\end{array}
$$

Table 2.2: Group Theory

be very effective with particular rewrite systems. In the present research we did not experiment with other possibilities, so the rest is left for future work[1].

2.3 Some Rewriting Systems

In this section we present a few sample rewriting systems for testing the algorithm. The first example is Group Theory, and the associated rewriting system that we used is shown in Table 2.2.

The second rewriting system involves groups in which all elements are of order 2, i.e. groups where $x^2 = id$ holds for all elements. we already presented a rewriting system for this theory, but we repeat it here in Table 2.24.

The third theory is yet another variation of Group Theory, where only the left identity and left inverse are axiomatized, and the right identity and right inverse axioms are proven. This is displayed in Table 2.4.

The fourth theory is AC and its associated rewriting system is given in Table 2.5.

The fifth theory is $\{\wedge, \vee, \neg\}$ Boolean Formulas, and the rewriting system, displayed in Table 2.6, is made so as to rewrite the formulas in Disjunctive Normal Form whenever possible.

[1]Please contact the othor for some preliminary results of expreimentation with rewrite systems where different rules are given different priorities.

$$
\begin{array}{rcl}
*(?x, *(?y, ?z)) & = & *(*(?x, ?y), ?z) \\
(*(?x, ?y), ?z) & = & *(?x, *(?y, ?z)) \\
(id, ?x) & \Rightarrow & ?x \\
(?x, id) & \Rightarrow & ?x \\
(inv(?x), ?x) & = & id \\
(?x, inv(?x)) & = & id \\
(?x, ?x) & = & id \\
?x & \Rightarrow & *(?x, *(?x, ?x))
\end{array}
$$

Table 2.3: Groups of elements of order 2

$$
\begin{array}{rcl}
((?x, ?y), ?z) & = & *(?x, *(?y, ?z)) \\
(id, ?x) & \Rightarrow & ?x \\
(inv(?x), ?x) & = & id \\
?x & \Rightarrow & *(*(inv(?x), ?x), ?x)
\end{array}
$$

Table 2.4: Group Theory, compact form

$$
\begin{array}{rcl}
+(+(x, y), z) & = & +(x, +(y, z)) \\
+(x, y) & = & +(y, x)
\end{array}
$$

Table 2.5: Associativity Commutativity

$$
\begin{array}{rcl}
\neg\neg x & \Rightarrow & x \\
\neg(x \wedge y) & \Rightarrow & \neg x \vee \neg y \\
\neg(x \vee y) & \Rightarrow & \neg x \wedge \neg y \\
(x \vee y) \wedge z & \Rightarrow & (x \wedge z) \vee (y \wedge z) \\
x \wedge (y \vee z) & \Rightarrow & (x \wedge y) \vee (x \wedge z) \\
(x \wedge y) \wedge z & = & x \wedge (y \wedge z) \\
(x \vee y) \vee z & = & z \vee (y \vee z) \\
x \vee \neg x & = & TRUE \\
x \wedge \neg x & = & FALSE \\
x \wedge y & = & y \wedge x \\
x \vee y & = & y \vee x \\
x \wedge TRUE & \Rightarrow & x \\
x \vee TRUE & = & TRUE \\
x \wedge FALSE & = & FALSE \\
x \vee FALSE & \Rightarrow & x \\
x \vee x & \Rightarrow & x \\
x \wedge x & \Rightarrow & x \\
\neg FALSE & \Rightarrow & TRUE \\
\neg TRUE & \Rightarrow & FALSE
\end{array}
$$

Table 2.6: $\{\wedge, \vee, \neg\}$ Boolean Formulas

2.4 Benchmark Problems

The implementation of the grammar rewriter was done in the programming language $C++$, and the details of the implementation are given in Appendix 1. In this section we present some benchmark problems relevant to the rewrite systems of the previous section. The results are given in tables that display statistics on the performance, for each different ordering strategy. The performance is evaluated using four measures. These are:

- **Rewrite steps:** These are the nontrivial rewrite steps that were needed, i.e. the rewrite steps that caused two different nonterminals to merge into one.
- **Rewrites produced:** This is the total number of rewrites that were inserted into the rewrite queue, at any point until the success of the proof.
- **Productions:** This is the total number of productions that were generated until the proof succeeded.
- **Substitution merges:** This is the total number of substitutions that were merged into one. The merging of substitutions was found to be the time bottleneck of the execution. Merging two substitutions involves, for each variable of the grammar[2], to check whether the two substitutions agree, or whether one is undefined. If this holds for all variables of the grammar, the two substitutions can be merged. The reader could get some idea of how fast the rewriter would run in his, by considering this operation performed as many times as indicated in the statistics below.

2.4.1 Problems involving Group Theory

The first problem involves the *Groups of elements of order 2* rewrite system. The problem is to prove that $x^2 = id$ implies that the group is abelian:

$$\begin{aligned} Let \quad & t = *(a,b) \\ Let \quad & s = *(b,a) \\ Show: \quad & t = s \end{aligned}$$

[2]Usually two to four variables

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	42	987	65	60171
LIFO	(*)	(*)	(*)	(*)
SIZE	37	772	44	44422
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	40	904	51	59140

Table 2.7: Group Theory, $x^2 = id$ implies *abelian*

The statistics for the running of the rewriter on this problem are given in Table 2.7. The first column exhibits the number of rewrite steps it took for the proof to complete. An (*) indicates that the program was stopped before completion, because much longer time was needed with these particular heuristics than with other heuristics. The second column indicates the total number of rewrites produced. The third column indicates the total number of productions produced. The fourth column indicates the total number of substitution merge operations performed.

The second problem involves *Group Theory* and is the left-cancellation law, namely $a * b = a * c$ implies $b = c$:

$$\begin{aligned} Let \quad & t = *(a, b) \\ let \quad & s = *(a, c) \\ Show: \quad & s = t \end{aligned}$$

The corresponding statistics for this problem are given in Table 2.8. This is a relatively hard problem, in which the $FIFO$ heuristics are slightly outperformed only by the $SIZE$ heuristics.

The third problem involves *Group Theory* and is another cancellation law, namely $b * a = b$ implies $a = id$:

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	98	2735	216	231220
LIFO	(*)	(*)	(*)	(*)
SIZE	88	2199	118	200691
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	(*)	(*)	(*)	(*)

Table 2.8: Group Theory, Left Cancellation Law

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	21	244	40	5774
LIFO	(*)	(*)	(*)	(*)
SIZE	15	197	27	4808
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	(*)	(*)	(*)	(*)

Table 2.9: Group Theory, Left Cancellation Law

$$\begin{aligned} \textit{Let} \quad & t = *(b, a) \\ \textit{Let} \quad & s = b \\ \textit{Let} \quad & s = t \\ \textit{Show} \quad & a = id \end{aligned}$$

The corresponding statistics for this problem are given in Table 2.9. This problem seems easier than the previous cancellation law, but the comparative results on the different heuristics are the same. Namely, $SIZE$ slightly outperforms $FIFO$ and every other strategy is much inferior.

The fourth problem involves *Group Theory* and is an exercise to prove that if $a * (b * c) = id$ then $(b * c) * a = id$:

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	129	4231	287	533975
LIFO	(*)	(*)	(*)	(*)
SIZE	195	8514	283	1602446
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	190	7180	332	1715077

Table 2.10: Group Theory, $a * (b * c) = id \Rightarrow (b * c) * a = id$

$$
\begin{aligned}
Let \quad & s = *(a, *(b, c)) \\
Let \quad & t = id \\
Let \quad & u = *(*(b, c), a) \\
let \quad & s = t \\
Show: \quad & u = t
\end{aligned}
$$

The corresponding statistics for this problem are given in Table 2.10. Here the $FIFO$ heuristics are better than any other heuristics.

The fifth problem, is to prove that the left identity and left inverse axioms alone, together with associativity, imply the whole group theory. So we use the compact form of group theory as the rewrite system.

$$
\begin{aligned}
Let \quad & s = a \\
Let \quad & t = *(a, id) \\
Show: \quad & s = t
\end{aligned}
$$

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	82	3004	176	402420
LIFO	(*)	(*)	(*)	(*)
SIZE	97	2489	116	178872
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	190	7180	332	1715077

Table 2.11: Group Theory, left identity and left inverse imply right identity

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	45	1024	106	65490
LIFO	(*)	(*)	(*)	(*)
SIZE	94	2372	113	162815
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	56	1761	106	121382

Table 2.12: Group Theory, left identity and left inverse imply right inverse

The statistics for this problem are given in Table 2.11, and again here the $FIFO$ heuristics perform best.

And the second part of the problem is:

$$\begin{aligned} Let \quad & s = id \\ Let \quad & t = *(a, inv(a)) \\ Show: \quad & s = t \end{aligned}$$

The statistics for this problem are given in Table 2.12, with similar results to the first part, except that now the $SIZE$ heuristics are worse than the $DISTANCE$ heuristics.

A simple problem in group theory, with many constants:

$$\begin{aligned} Let \quad & s = *(*(a,b), *(*(c, *(inv(c), inv(b))), inv(a))) \\ Let \quad & t = id \end{aligned}$$

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	68	1286	114	68318
LIFO	(*)	(*)	(*)	(*)
SIZE	34	492	54	16900
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	8	57	22	817

Table 2.13: Group Theory, easy identity

$$Show: \quad s = t$$

This problem is interesting because it can be proven by using only rules that contract terms, or associativity. In other words, the rewrite rule $x = *(*(inv(x), x), x)$ does not need to be used. The statistics for this problem are given in Table 2.13. It is interesting to note here the superiority of the $DISTANCE$ heuristics. Below is the proof produced by running with these heuristics:

1. $c * (inv(c) * inv(b)) \rightarrow (c * inv(c)) * inv(b)$
2. $c * inv(c) \rightarrow id$
3. $id * inv(b) \rightarrow inv(b)$
4. $(a * b) * (inv(b) * inv(a)) \rightarrow ((a * b) * inv(b)) * inv(a)$
5. $(a * b) * inv(b) \rightarrow a * (b * inv(b))$
6. $b * inv(b) \rightarrow id$
7. $a * id \rightarrow a$
8. $a * inv(a) \rightarrow id$

The reason is that all the Rewrites needed for this proof are of zero weight according to the $Weight_{distance}$ function. It is safe to say that the $DISTANCE$ heuristics is good for exactly this type of problems.

A last problem about group theory, involves *homomorphisms*. For that reason, we add a new axiom in the rewrite system of group theory:

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	16	267	46	7859
LIFO	(*)	(*)	(*)	(*)
SIZE	23	303	31	9261
CONTEXT	10	120	18	2058
DISTANCE	14	158	22	3993

Table 2.14: Group Theory, a homomorphism carries identity to identity

Axiom for a homomorphism ϕ:

$$\phi(*(?x, ?y)) \quad = \quad *(\phi(?x), \phi(?y))$$

The problem is the following:

$$\begin{aligned} \mathit{Let} \quad & s \; = \; \mathit{id} \\ \mathit{Let} \quad & t \; = \; \phi(\mathit{id}) \\ \mathit{Show} \quad & s \; = \; t \end{aligned}$$

Namely that a homomorphism carries identity to identity. The corresponding statistics are given in Table 2.14. We included this example as a demonstration that the $CONTEXT$ heuristics do in fact outperform the other heuristics in some cases in the context of group theory, and in a problem that is not artificially produced.

2.4.2 Problems involving Logical Expressions

In this section we present statistics from problems involving logical expressions, that we run using our *Boolean Formulas* rewrite system. These problems are taken from various papers in which they

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	202	2632	285	422622
LIFO	(*)	(*)	(*)	(*)
SIZE	(*)	(*)	(*)	(*)
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	53	392	93	19081

Table 2.15: Boolean Formulas, $q \rightarrow ((p \vee q) \wedge (\neg p \vee q))$

were used as benchmark problems. In this section we will drop the prefix notation on the terms. In order to write these problems in a form readeble by our rewriting system, we used the following transformation: a subformula of the form $A \rightarrow B$ is translated to the subformula $\neg A \vee B$.

The first problem is taken from [7]. It is to prove the identity $q \rightarrow ((p \vee q) \wedge (\neg p \vee q))$. It is expressed in the rewriting system as follows:

$$\begin{aligned} \mathit{Let} \quad s &= \neg q \vee ((p \vee q) \wedge (\neg p \vee q)) \\ \mathit{Let} \quad t &= \mathit{TRUE} \\ \mathit{Show} \quad s &= t \end{aligned}$$

The statistics for this problem are displayed in Table 2.15. In this problem, $DISTANCE$ heuristics clearly outperform all other heuristics.

The next problem is taken from [7]. It is to prove the identity $q \rightarrow ((p \vee q) \wedge (\neg p \vee q))$. The statistics for this problem are displayed in Table 2.16. It is expressed in the rewriting system as follows:

$$\mathit{Show} \quad ((p \vee q) \wedge \neg q) \wedge (\neg p \wedge (\neg p \vee \neg q)) = \mathit{FALSE}$$

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	188	3506	365	8581704
LIFO	(*)	(*)	(*)	(*)
SIZE	46	452	70	82445
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	102	1453	205	5619922

Table 2.16: Boolean formulas, $q \rightarrow ((p \vee q) \wedge (\neg p \vee q))$

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	9	34	22	314
LIFO	17	69	36	1469
SIZE	7	27	17	272
CONTEXT	17	65	34	1006
DISTANCE	6	24	16	210

Table 2.17: Boolean Formulas, $(\neg q \rightarrow \neg p) \rightarrow (p \rightarrow q)$

There follow some problems taken from [11], which is a paper with benchmark problems for equality theorem provers. In this paper, points are given to each problem for difficulty. The first problem is to prove the identity $(\neg q \rightarrow \neg p) \rightarrow (p \rightarrow q)$. It is given 2 points of difficulty. The statistics are found in Table 2.17, and it is expressed in the rewriter as follows:

$$\begin{aligned} \mathit{Let} \quad s &= \neg(\neg p \vee q) \vee (\neg\neg q \vee \neg p) \\ \mathit{Let} \quad t &= \mathit{TRUE} \\ \mathit{Shows} &= t \end{aligned}$$

The next problem is "a biconditional verison of the most difficult theorem proved by the new logic theorist (Newell, 1972)", found in [11]. It is to prove the identity $(\neg\neg p \rightarrow p) \wedge (p \rightarrow \neg\neg p)$. It is

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	4	36	13	707
LIFO	(*)	(*)	(*)	(*)
SIZE	3	24	10	272
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	4	29	11	387

Table 2.18: Boolean Formulas, $(\neg\neg p \rightarrow p) \wedge (p \rightarrow \neg\neg p)$

given 2 points of difficulty. The statistics for this problem are given in Table 2.18, and the $SIZE$ heuristics is the best for this problem. It is expressed as follows:

$$\begin{aligned} Let \quad & s = (\neg\neg\neg p \vee p) \wedge (\neg p \vee \neg\neg p) \\ let \quad & t = TRUE \\ Show \quad & s = t \end{aligned}$$

The next problem is "the hardest theorem proved by a breadth-first logic theorist (Siklossy et al. 1973)" [11]. It is to prove the identity $\neg(p \rightarrow q) \rightarrow (q \rightarrow p)$. It is expressed as follows:

$$\begin{aligned} Let \quad & s = (\neg\neg\neg p \vee p) \wedge (\neg p \vee \neg\neg p) \\ Let \quad & t = TRUE \\ Show \quad & s = t \end{aligned}$$

This problem is given 1 point of difficulty. The statistics are given in Table 2.19. Notice here that

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	62	819	100	182495
LIFO	(*)	(*)	(*)	(*)
SIZE	17	80	32	1958
CONTEXT	145	647	189	23111
DISTANCE	13	74	27	2055

Table 2.19: Boolean Formulas, $\neg(p \rightarrow q) \rightarrow (q \rightarrow p)$

CONTEXT heuristics outperform *FIFO* heuristics in terms of time, although *FIFO* heuristics find a shorter proof.

The next problem is given 2 points of difficulty in [11], and it is to prove the identity $(\neg p \rightarrow q) \rightarrow (\neg q \rightarrow p)$. It is expressed in the rewriter as follows:

$$\begin{aligned} let \quad & s = \neg(\neg\neg p \vee q) \vee (\neg\neg q \vee p) \\ Let \quad & t = TRUE \\ Show \quad & s = t \end{aligned}$$

The statistics for this problem are given in Table 2.20. We notice here that the *SIZE* and *DISTANCE* heuristics coincide to find the same proof. The reason is that all steps in the proof are of the same "distance" from the goal, and in the case of ties, *DISTANCE* heuristics is designed to use *SIZE* heuristics.

The next problem is given 5 points of difficulty in [11]. It is to prove the identity $((p \rightarrow q) \rightarrow p) \rightarrow p$. It is expressed in the rewriter as follows:

$$\begin{aligned} let \quad & s = \neg(\neg(\neg p \vee q) \vee p) \vee p \\ Let \quad & t = TRUE \end{aligned}$$

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	8	31	21	347
LIFO	19	77	39	1891
SIZE	7	28	17	280
CONTEXT	18	74	39	1702
DISTANCE	7	28	17	280

Table 2.20: Boolean Formulas, $(\neg p \rightarrow q) \rightarrow (\neg q \rightarrow p)$

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	35	197	55	5620
LIFO	(*)	(*)	(*)	(*)
SIZE	23	115	38	2152
CONTEXT	(*)	(*)	(*)	(*)
DISTANCE	23	107	45	2608

Table 2.21: Boolean Formulas, $((p \rightarrow q) \rightarrow p) \rightarrow p$

$$Show \quad s = t$$

The statistics for this problem are given in Table 2.21.

The next problem is to prove the identity $(p \rightarrow q) \vee (q \rightarrow p)$. It is expressed in the rewriter as follows:

$$let \quad s = (\neg p \vee q) \vee (negq \vee p)$$
$$Let \quad t = TRUE$$
$$Show \quad s = t$$

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	60	847	98	222678
LIFO	(*)	(*)	(*)	(*)
SIZE	10	54	21	1445
CONTEXT	145	636	179	21022
DISTANCE	10	50	22	1502

Table 2.22: Boolean Formulas, $(p \rightarrow q) \vee (q \rightarrow p)$

The statistics for this problem are given in Table 2.22.

2.4.3 Associativity Commutativity

The last problem is from AC, where any strategy provably succeeds in a bounded number of steps. This is true because all equivalence classes of ground terms are of bounded cardinality. The problem is an example of an equivalence between two terms with five different constants each:

$$\begin{aligned} \mathit{let} \quad s &= +(d, +(+(a, +(e, b)), c)) \\ \mathit{Let} \quad t &= +(+(b, d), +(e, +(c, a))) \\ \mathit{Show} \quad s &= t \end{aligned}$$

Strangely enough, the $FIFO$ ordering outperforms all other heuristics.

Heuristics	Steps	Rewrites	Productions	Subst-Merges
FIFO	46	443	78	3229
LIFO	1441	10914	1712	60918
SIZE	100	946	128	6357
CONTEXT	482	3189	655	9614
DISTANCE	102	987	130	6479

Table 2.23: Associativity Commutativity, on a term with five different constants

Conclusions

Implementing an equality theorem prover is a challenging programming task. Once the theorem prover was implemented, we aimed at exploring its possibilities by considering various strategies for ordering the queue of rewrites. We managed to solve several standard benchmark problems from the literature of term rewriting, by using only general heuristics, that are not hardwired for high performance in any particular equation theory. We ran a total of 17 benchmark problems.

Table 2.1 summarizes the results of this testing. It shows how many problems each heuristics solved, depending on the number of steps that was put as an upper bound. Notice that $LIFO$ solved an additional problem in 1441 steps, and $CONTEXT$ solved the same problem in 482 steps. But in general we stopped the program in much fewer steps, because the time it takes to complete step n increases with n, rendering testing cumbersome.

A few remarks on the statistics: The three heuristics that perform best, are the $FIFO$, $DISTANCE$, and $SIZE$ heuristics. In many problems, especially the "easy" problems, $DISTANCE$ or $SIZE$ are best. Depending on the problem, they outperform $FIFO$ by a considerable factor, and find nearly "shortest" proofs to the equation theorem. But $FIFO$ seems to be the most robust choice for harder problems.

One first thing to point out, is that the heuristics that performed best, were those that satisfied fairness. Although there are problems for which $CONTEXT$, which is not fair, performs better than any other heuristics, it is safe to say that in general all fair strategies outperformed all the non-fair strategies.

Intuitively, one might expect that the best heuristics would be $DISTANCE$. This heuristics employs a notion of "distance" of a rewrite from the goal, depending on the context and size of the lhs of the rewrite. However, the results show that $FIFO$ is more robust. A closer look on the particular examples where $DISTANCE$ performs best, or worst, gives an explanation: $DISTANCE$ performed best in the easy identity of group theory, namely to prove that $a * b * c * c^{-1} * b^{-1} * a^{-1}$ equals identity. This is an obvious equality for humans, but $CONTEXT$ and $LIFO$ failed, whereas $FIFO$ required 68 nontrivial union operations. $DISTANCE$ on the other hand, found a nearly shortest proof. The reason is that distance performs first all the rewrites of subterms of the goal, and of terms that are proven to be equal to subterms of the goal, and no bigger than these subterms.

Thus it is good in performing manipulations of the terms that are easy for us to see automatically. $DISTANCE$ is closer than other heuristics to our notion of an obvious identity. On the other hand, $DISTANCE$ failed in the left cancellation identities. The reason is that these proofs require "seeing" some auxiliary terms. Specifically, let's have a look at my proof of $a * b = a * c \Rightarrow b = c$:

$a * b = a * c \Rightarrow$

$a^{-1} * (a * b) = a^{-1} * (a * c) \Rightarrow$

$(a^{-1} * a) * b = (a^{-1} * a) * c \Rightarrow$

$id * b = id * c \Rightarrow$

$b = c$

This proof requires to "see" (i.e. intern) the term $a^{-1} * (a * b)$. $DISTANCE$ will give the lowest priority to such an interning. Specifically, the interning requires to see that $id = a * a^{-1}$ (or, translated in the group theory rewrite system, that $a = a * (a * a^{-1})$). But a is not a subterm of the goal, and so this will never take place because the corresponding rewrites will always have lowest priority. So, in this proof, and all similar proofs that require the introduction of auxiliary terms that are not subterms of the goal, $DISTANCE$ diverges. Future work would involve modification of the notion of goal, and of the notion of fairness for a grammar rewriting weight heuristics, to accommodate for this deficiency. Still, as such $DISTANCE$ works very well for problems where all the relevant terms are subterms of the terms in the goal, and it had the best performance in 7/17 of the benchmark problems. $SIZE$ had the best performance in 5/17 of the problems (one tie with $DISTANCE$), $FIFO$ in 4/17, and $CONTEXT$ in 1/17.

Concluding, we would like to point out that it is still an open problem to determine the power of Grammar Rewriting when used with other heuristic strategies besides the ones we defined, and when used in the more general context of induction theorem proving. Also, it is an open problem to determine the performance of Grammar Rewriting based special purpose theorem provers, for example for logical expressions. Future work in this area could go to either of these directions.

Appendix 1: The implementation

For the implementation of the rewriter we used the programming language $C++$, Classes in $C++$ were used to represent the objects of our system. In particular, we defined the following classes, for the corresponding mathematical objects. These structures will be described in more detail in the next sections:

- **Typed List** denoted by (Obj_1, ..., Obj_n) where for $1 < i < n$ Obj_i is an object of the type of the list. If $n = 0$, then the list is denoted by either () or NIL.
- **Symbol** denoted by its name, which is a character string. If it is a variable, then '?' is added before its name.
- **Term** denoted by $f(subterm_1, \ldots, subterm_n)$ where f is a function symbol and for $1 < i < n$ $subterm_i$ is a term. The parentheses can be omitted, if $n = 0$.
- **Nonterminal** denoted by X, where X is a capital letter.
- **Production** denoted by $X \rightarrow f(Y_1, ..., Y_n)$ where X and Y_i for $1 < i < n$ are nonterminals, and f is a function symbol.
- **Pattern** denoted by $f(subpattern_1, \ldots, subpattern_n)$ where f is a symbol, and if f is a variable then $n = 0$, else $n \geq 0$ and for $1 < i < n$ $subpattern_i$ is a pattern.
- **Match** denoted by $< X \ . \ p \ . \ slist >$ where X is a nonterminal, p is a pattern, and $slist$ is a list of substitutions.
- **Substitution** denoted by $\{[X_1] \ \ldots \ [X_n]\}$ where n is the number of variables used in the problem instance, and X_i for $1 < i < n$ is either a nonterminal, or $*$, so that the variable i is carried to nonterminal X_i, or nowhere in the case of $*$.
- **Rewrite** denoted by $t \rightarrow s$ where s, t are terms. Alternatively, it is denoted as $rewrite(X, L \rightarrow R, \sigma)$ where X is a nonterminal, L, R are patterns, $L \rightarrow R$ is a rewrite rule, σ is a substitution, and X matches L under σ.
- **Heap**
- **Grammar**

Symbols, Terms, and Patterns

A **Symbol** is a tagged character string. It consists of a *name*, which is a character string, and a *tag*, which can be one of { $VARIABLE$, $FUNCTION\ SYMBOL$ }. Symbols are stored in a hash table, so that every symbol exists in only one copy in the memory. The hash table of symbols is an array of linked lists of symbols, so that its size can be assumed to be unbounded, and its access time can be assumed to be constant on average.

A **Term** is either:

- **A leaf term** that consists of a symbol of tag $FUNCTION\ SYMBOL$ and a NIL list of subterms. This is the way a constant is represented.
- **A function application** that consists of a symbol of tag $FUNCTION\ SYMBOL$ and a non-empty list of subterms.

A term is represented as follows: it contains pointer to a symbol the $prefix$, an array of pointers to subterms, and an integer *size* denoting the size of the subterm array. It also contains a pointer to a Pattern, which is $NULL$ if the term does not match any pattern, and points to the pattern that the term matches, it there is one. In that way, tha pattern that a particular term matches to, can be accessed in constant time. A restriction for terms is that there is exactly one term that points to a particular function symbol. Hence, the *arity* (number of sybterms) of every symbol is uniquely defined. Terms are stored in a hash table of terms, that is an array of linked lists of terms, with unbounded size and constant average access time. Every term exists in only one copy in the memory.

A **Pattern** is either:

- **A leaf pattern** that consists of a symbol of arbitrary tag, and a NIL list of subterms. So, this is either a constant pattern, or a variable pattern.
- **A function application pattern** that consists of a symbol of tag $FUNCTION\ SYMBOL$ and a non-empty list of subpatterns.

A pattern is represented as follows: it contains a pointer to a symbol, an array of pointers to subpatterns, and an integer *size* denoting the size of the subpattern array. it also contains a pointer to a list of terms, that contains the terms that match this particular pattern. A restriction for

patterns is that there is at most one pattern that points to a particular variable or function symbol. The arity of the function symbol for all patterns except variables, is required to be the same as the arity of the corresponding term that points to the same function symbol. Patterns are stored in a hash table of patterns, that is an array of linked lists of patterns, with unbounded size and constant average access time. Every pattern exists in only one copy in the memory.

A term $t = f(subterm_1, ..., subterm_n)$ **matches** a pattern $p = g(subpattern_1, \ldots, subpattern_n)$ if the following hold:

- $n = 0$, and either $f = g$ or g is a symbol of tag $VARIABLE$, or
- $n > 0$, and $f = g$, and for $1 < i < n$ $subterm_i matches subpattern_i$

We say that a variable $?x$ **appears** in a pattern $p = f(q_1, ..., q_n)$, if:

- $f = ?x$, or
- for some i s.t. $1 < i < n$, $?x$ appears in q_i

Now we are ready to define the language that we use to write a problem statement in a form readable by our system.

Expressing a Problem Statement

A simple language for expressing a problem statement is defined. The code written in this language should be contained in a file devoted to the definition of this problem. The language consists of terms, that are constructed using the following special symbols: $NEW-GRAMMAR$, $CALL-UNION$, $INTERN$, $REWRITE-RULES$, $EQUAL$, $PROVE-EQUAL$, $CONTEXT-HEURISTICS$, $SIZE-HEURISTICS$.

A problem statement is written by means of the following grammar:

$< problem - statement > ::= < new - grammar > < listof - commands > < problem >$

$< new - grammar > ::=$ **NEW-GRAMMAR**$(< grammar - name >, < listof - equalities >, < heuristics >)$

$< listof - commands > ::= NIL \mid$
$< assumption > < listof - commands > \mid$
$< binding > < listov - commands >$

$< assumption >$ ::= **CALL-UNION**($< term >_1, < term >_2$)

$< binding >$ ::= **INTERN**($< grammar - name >, < term >_1, < term >_2$) where $< grammar - name >$ appears in a previous $< problem - statement >$ and $< term >_2$ is a constant.

$< listof - equalities >$ ::= $NIL \mid < equality >, < listof - equalities >$

$< equality >$ ::= **EQUAL**($< pattern_1 >, < pattern_2 >$) where every variable that appears in $< pattern_2 >$ appears also in $< pattern_1 >$

$< heuristics >$ ::= **CONTEXT-HEURISTICS | SIZE-HEURISTICS**

$< problem >$ ::= **PROVE-EQUAL**($< term >_1, < term >_2$) where $< term >_1$ and $< term >_2$ are constants

where $< grammar - name >$ is a symbol, $< term >_i$ is a term for any i, and $< pattern >_i$ is a pattern for any i. Under these definitions, it is clear that $< problem - statement >$ produces a sequence of terms, where each of $< new - grammar >$, *assumption*, $< equality >$, $< heuristics >$, and $< problem >$ produce terms. in the $< problem >$ production, $< term >_i$ for $i = 1, 2$ is assumed to be "bound" to a term, by a previous $INTERN$ command.

We give an example below, to illustrate the use of the above language:

Say we want to express a problem in group theory, and in particular to prove that for a group G, if for all $x \in G$ $x^2 = identity$, then G is abelian.

The axioms of group theory together with $x^2 = x$ are the following, written in the form of equalities: (here $*$ denotes group composition, id denotes the identity element, and inv denotes the inverse function).

$$\begin{aligned} x * (y * z) &= (x * y) * z \\ id * x &= x \\ x * id &= x \\ x * inv(x) &= x \\ inv(x) * x &= x \\ x * x &= id \end{aligned}$$

The above axiom set can be expressed by a rewrite system using our language, as follows:

$$
\begin{aligned}
*(?x, *(?y, ?z)) &= *(*(?x, ?y), ?z) \\
(*(?x, ?y), ?z) &= *(?x, *(?y, ?z)) \\
(id, ?x) &= ?x \\
(?x, id) &= ?x \\
(inv(?x), ?x) &= id \\
(?x, inv(?x)) &= id \\
(?x, ?x) &= id \\
?x &= *(?x, *(?x, ?x))
\end{aligned}
$$

Table 2.24: Groups of elements of order 2

NEW-GRAMMAR($groups - of - elts - of - order - 2$,
EQUAL$(*(?x, *(?y, ?z)), *(*(?x, ?y), ?z))$,
EQUAL$(*(*(?x, ?y), ?z), *(?x, *(?y, ?z)))$,
EQUAL$(*(?x, id), ?x)$,
EQUAL$(*(id, ?x), ?x)$,
EQUAL$(*(?x, inv(?x)), id)$,
EQUAL$(*(inv(?x), ?x), id)$,
EQUAL$(*(?x, ?x), id)$,
EQUAL$(?x, *(?x, *(?x, ?x)))$,
CONTEXT-HEURISTICS)

Some notational conventions are useful. From now on, instead of expressing a rewrite system in the above machine-readable form, we will express it as a sequence of ordered equalities, where the left-hand side rewrites to the right-hand side. According to this convention, the above rewrite system will be expressed as in Table 2.24:

Resuming to our demonstration of how to express the problem statement:

Our goal is to prove that the above axioms imply that the commutative law holds in the group. If we show that the commutative law holds for two *generic* (i.e. arbitrary) constants in the group, then we are done. We express that with the three commands below:

INTERN(groups-of-elts-of-order-2, $*(a, b)$, t)

INTERN(groups-of-elts-of-order-2, $*(b, a)$, s)
PROVE-EQUAL(groups-of-elts-of-order-2, t, s)

Nonterminals, Productions, Substitutions, and Matches

A **Nonterminal** is identified with an integer, which is its *identity number*. It is represented as follows: It contains its identity number, a list of pointers to all the productions that come from it, a list of pointers to all the productions in whose right-hand side it appears, and a hash table of all the matches that apply to it. For the purposes of the *Heap* heuristics, it also contains a set of integer weights, in particular two integer weights in our implementation. Those are the *size*, and the *context − size* of it. For the purpose of expressing a grammar and the associated rewrites in a convenient readable way, a nonterminal also contains a pointer to the *minimal* production coming from it. The quantities *size*, *context − size* and *minimal production* of a nonterminal will be defined in the following chapter, on the rewrite queue ordering heuristics. A nonterminal also contains a pointer to a nonterminal, the $FIND$ pointer. Denote the $FIND$ pointer nonterminal of a nonterminal X by $X \rightarrow FIND$. When two nonterminals are merged into one by a rewriting that takes place, then one of the two nonterminals "dies", and its $FIND$ pointer points to the nonterminal that "survives". Thus, a nonterminal is *alive* when its $FIND$ pointer is $NULL$, and *dead* otherwise. Because a nonterminal contains a hash table of all the matches that apply to it, we can find in constant time all the matches of a nonterminal to a specific pattern.

We say that a nonterminal X **produces** a term $t = f(s_1, ..., s_n)$ iff:

- $n = 0$ and $X \rightarrow f()$ is a production of the grammar, or
- $n > 0$ and $\exists$ a production $X \rightarrow f(Y_1, ..., Y_n)$ s.t. $\forall i : 1 < i < n$, Y_i produces s_i.

A **Production** $X \rightarrow f(Y_1, ..., Y_n)$ is represented as follows: It contains a pointer to the nonterminal X, a pointer to the function symbol f, an array of pointers to the nonterminals Y_i for $1 < i < n$, and an integer *size* that equals n. Productions are kept in a hash table that is an array of lists of productions, so that a particular production can be accessed in average in constant time. A production also contains a tag $LIFE$ that is $TRUE$ if the production is *alive*, and $FALSE$ if the production is *dead*. A production is *dead* if X is *dead*, or for some $1 < i < n$ Y_i is *dead*.

A **Substitution** σ is an ordered set of nonterminal values, where a *nonterminal value* is either a nonterminal, or NIL. Thus $\sigma = \{[X_1]...[X_n]\}$. The *length* of a substitution is the integer n. In a

particular grammar, all the substitutions have the same *length*, that equals the number of variables that *appear* in the0 $< problem - statement >$. A substitution is represented as follows: it contains its *length*, plus an array of pointers to nonterminals, the *bindings* array. The substitutions are held in a hash table of substitutions. This hash table is represented as an array of lists of substitutions, so that we can assume that it has unbounded size and average constant access time.

A substitution $\sigma = \{[X_1]...[X_n]\}$ is *valid*, if for all $1 < i < n$ X_i is *alive*. Two substitutions $\sigma = \{[X_1]...[X_n]\}$ and $\tau = \{[Y_1]...[Y_n]\}$ are equal $\sigma = \tau$, iff for all $1 < i < n$, $X_i \rightarrow FIND = NULL$ and $X_i = Y_i$, or $X_i \rightarrow FIND = Y_i \rightarrow FIND \neq NULL$. A substitution $\sigma = \{[X_1]...[X_n]\}$ is *smaller* than a substitution $\tau = \{[Y_1]...[Y_n]\}$, denoted by $\sigma \leq tau$ iff for all $1 < i < n$, the *nonterminal value* $[X_i]$ is $\leq$ the *nonterminal value* $[Y - i]$, meaning that either $[X_i] = NULL$, or $X_i \rightarrow FIND = NULL$ and $X_i = Y_i$, or $X_i \rightarrow FIND = Y_i \rightarrow FIND \neq NULL$.

A **Match** $\mu = < X \ . \ p \ . \ (\sigma_1, ..., \sigma_n) >$ is represented as follows: it contains a pointer to the nonterminal X, a pointer to the pattern p, and a list of pointers to the substitutions σ_i for $1 < i < n$. Thus, a match object in the implementation, contains arbitrarily many matches as defined in the previous chapter, one for each substitution σ_i that is different on some relevant variable. The matches of a nonterminal X are kept in a hash table specific to the nonterminal X. This hash table is implemented as an array of lists of pointers to matches, so that it can be assumed to have unbounded size and on average constant access time.

Monitoring the rewriter

We defined a simple language for interactively monitoring the rewriting process after a problem has been specified. In this section we will briefly describe this language. The special symbols of this language are: $MATCHES - OF$, $NONTERMINALS$, $PRODUCTIONS$, $VARIABLES$, $PATTERNS$, $REWRITES$, $STOP$, $CLEANUP$.

The program is run as follows: an argument file that contains the problem statement is given to the executable. After reading the problem statement, the program stops and waits for a command. The special command $STOP$, followed by an integer n, runs the rewriter till success, or till the rewrite queue is empty, or till n proper nonterminal merges have occurred. Then it resumes to the interactive mode. The remaining of the commands are explained below:

- The command $VARIABLES$ prints all the variables of the problem statement.

- The command $PATTERNS$ prints all the patterns of the problem statement.
- The command $NONTERMINALS$ prints all the alive nonterminals of the grammar so far, each together with the minimal term that is produced by the grammar, starting at this nonterminal.
- The command $PRODUCTIONS$ prints all the alive productions of the grammar so far.
- The command $MATCHES - OF(X)$ where X is a nonterminal in the grammar, prints all the matches of this nonterminal.
- The command $REWRITES$ prints the rewrite queue, as it is ordered at this moment.
- The command $CLEANUP$ exists for optimization purposes, and cleans up the grammar by removing all *dead* substitutions, nonterminals, and matches. Such a cleanup is performed by default periodically at a prespecified rate.

prompt> **rewriter groups-of-elts-of-order-2**

reading problem statement ... done

> **STOP**

> **5**

1. $a \quad \Rightarrow \quad *(a, *(a, a))$
2. $b \quad \Rightarrow \quad *(b, *(b, b))$
3. $*(b, a) \quad \Rightarrow \quad *(*(b, a), *(*(b, a), *(b, a)))$
4. $*(a, b) \quad \Rightarrow \quad *(*(a, b), *(*(a, b), *(a, b)))$
5. $*(a, a) \quad \Rightarrow \quad id$

> **NONTERMINALS**

$A \quad \rightarrow \quad a$

$B \quad \rightarrow \quad b$

$C \quad \rightarrow \quad *(a, b)$

$D \quad \rightarrow \quad *(b, a)$

$E \quad \rightarrow \quad id$

$G \quad \rightarrow \quad *(b, b)$

$I \quad \rightarrow \quad *(*(b, a), (b, a))$

$K \quad \rightarrow \quad *(*(a, b), *(a, b))$

> **PRODUCTIONS**

$A \rightarrow a()$

$A \rightarrow *(A, E)$

$B \rightarrow b()$

$B \rightarrow *(B, G)$

$C \rightarrow *(A, B)$

$C \rightarrow *(C, K)$

$D \rightarrow *(B, A)$

$D \rightarrow *(D, I)$

$E \rightarrow *(A, A)$

$E \rightarrow id$

$G \rightarrow *(B, B)$

$I \rightarrow *(D, D)$

$K \rightarrow *(C, C)$

> MATCHES-OF(A)

$< A \ . \ *(?x, *(?y, ?z)) \ . \ (\{[A][A][A]\}) >$

$< A \ . \ ?x \ . \ (\{[*][*][A]\}) >$

$< A \ . \ *(?y, ?z) \ . \ (\{[E][A][*]\}) >$

$< A \ . \ ?y \ . \ (\{[*][A][*]\}) >$

$< A \ . \ *(?x, *(?x, ?x)) \ . \ (\{[*][*][A]\}) >$

$< A \ . \ *(?x, id) \ . \ (\{[*][*][A]\}) >$

$< A \ . \ ?z \ . \ (\{[A][*][*]\}) >$

$< A \ . \ *(*(?x, ?y), ?z) \ . \ (\{[E][E][A]\}) >$

$< A \ . \ *(?x, ?y) \ . \ (\{[*][E][A]\}) >$

> CLEANUP

done cleaning up

> STOP

And so forth. The above proof succeeds within 44 rewrite steps.

Rewrites, the Heap, and the Grammar

A **Rewrite** $r = rewrite(X, L \rightarrow R, \sigma)$ is represented as follows: It contains a pointer to the non-

terminal X, a pointer to the pattern L and one to th pattern R, and a pointer to the substitution σ. It contains an integer *idno* which is its "ID number", indicating how many rewrites were constructed before r. It also contains the integers $Key_{distance}$, $Key_{context}$, and Key_{size}, that are used in heuristics that we are going to explain later. Rewrites are held in a heap. A **heap** for rewrites is an array of pointers to rewrites. It contains an integer *length* that indicates the number of rewrites in the heap, and has a maximum size $LMAX$ [3]. It also contains a tag *HEURISTIC* that indicates the heuristics chosen. It is a binary heap, whereby each node has a parent and two children. This data structure is described in detail in [6], and we are basing our implementation in the description given there. Below we explain exactly how this data structure works.

The array of rewrites, call it A, is arranged as follows: The largest element is in position $A[1]$, and for each $1 \leq i \leq length$, the children of $A[i]$ are $A[2i]$ and $A[2i+1]$. The following invariant holds in a binary heap: the two children of a node have smaller key than their parent.

- *Insert* taking one argument, namely the rewrite r to be inserted.
- *Extract-Max* that extracts the element with the maximum key.

The invariant holds in the beginning and in the end of each of the two interface functions. There are also some internal functions, that are used as The heap supports the following interface functions: building blocks for the implementation of the external functions. These internal functions are:

- **parent**, taking one argument, an integer i, and returning $floor(i/2)$.
- **left**, **right**, taking one argument, i, and returning $2i$ and $2i+1$ respectively.
- **key**, taking one argument, i, and returning the key of $A[i]$ according to the current heuristics.
- **larger-key**, taking i, j as arguments, and returning $TRUE$ if $key(i) \leq key(j)$ and $FALSE$ otherwise.
- **exchange**, taking i, j as arguments, and exchanging the positions of $A[i]$ and $A[j]$.
- **heapify**, taking one argument, i, and ensuring the heap invariant for the subtree that starts at position $A[i]$.

[3] Large enough for our purposes.

- **tide-up**, taking the grammar class object as an argument, and recomputing all the keys of the rewrites, to be updated according to the grammar. This is called once after a rewrite is performed, to keep the keys always updated.

Below are the algorithms for performing the nontrivial of the above functions, namely Insert, Extract-Max, and heapify:

- **Insert**(r)
  1. Increase the *length* by 1
  2. Put r in the end of the heap
  3. Say r is $A[i]$. While $i > 1$ and $key(parent(i)) > key(i)$, do
     (a) $exchange(parent(i), i)$
     (b) $i = parent(i)$
- **Extract-Max**
  1. $exchange(1, length)$
  2. Decrease the *length* by 1
  3. $heapify(1)$
  4. Return the old $A[1]$
- **heapify**(i)
  1. Let K_i be the key of $A[i]$, K_l be the key of $A[left(i)]$, and K_r be the key of $A[right(i)]$.
  2. Let $K_{largest} = maxK_i, K_l, K_r$, and *largest* be the corresponding index.
  3. If $K_{largest} \neq K_i$ then do
     (a) $exchange(i, largest)$
     (b) $heapify(largest)$

  The running time of $heapify$ is $O(log(n))$, where n is the size of the array A.

For a more detailed discussion on the heap data structure, we refer to [6], from which book we got the algorithm. Below is the algorithm for *larger-key*, which determines which of the two keys $key(i)$ and $key(j)$ is larger. This is done according to the current heuristics. A tag $HEURISTICS$ is held in the heap, which determines the current heuristics. Then, *larger-key* operates as follows:

- **larger-key**(i, j)
  - Switch according to $HEURISTICS$:
    1. Case $HEURISTICS = FIFO$
       (a) If $index(A[i]) < index(A[j])$ return $TRUE$
       (b) else return $FALSE$
    2. Case $HEURISTICS = LIFO$
       (a) If $index(A[i]) > index(A[j])$ return $TRUE$
       (b) else return $FALSE$
    3. Case $HEURISTICS = SIZE$
       (a) If $Key_{size}(A[i]) < Key_{size}(A[j])$ return $TRUE$
       (b) else if $Key_{size}(A[j]) < Key_{size}(A[i])$ return $FALSE$
       (c) else use $FIFO$ heuristics
    4. Case $HEURISTICS = DISTANCE$
       (a) If $Key_{distance}(A[i]) < Key_{distance}(A[j])$ return $TRUE$
       (b) else if $Key_{distance}(A[j]) < Key_{distance}(A[i])$ return $FALSE$
       (c) else use $SIZE$ heuristics
    5. Case $HEURISTICS = CONTEXT$
       (a) If $Key_{context}(A[i]) < Key_{context}(A[j])$ return $TRUE$
       (b) else if $Key_{context}(A[j]) < Key_{context}(A[i])$ return $FALSE$
       (c) else use $DISTANCE$ heuristics

The **Grammar** class, is a storage class, that contains in hash tables all the nonterminals, productions, and matches of a particular grammar[4]. Then, on these objects, the algorithm given in Chapter 1 is performed, using the queue heuristics described in Chapter 3, and implemented as explained above.

[4]As we said before, the matches are contained in hash tables inside their nonterminals, and not actually in the grammar class.

Bibliography

[1] Franz Baader. Rewrite systems for varieties of semigroups. *CADE*, 10:381–395, 1990.

[2] Robert S. Boyer and Struther Moore. A computational logic. *ACM Monograph Series, Academic Press*, 1979.

[3] L. Bachmair N. Dershowitz and D. Plaisted. Completion without failure. *Proc. Col on Resolution of Equations in Algegraic Structures*, 1987.

[4] J. Hsaing and M. Rusinowitch. On word problems in equational theories. *ICALP-87, LCNS 267*, pages 54–71, 1987.

[5] Leslie P. JChew. An improved algorithm for computing with equations. *FOCS80, IEEE Computer Society Press*, pages 108–117, 1980.

[6] Thomas H. Cormen Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge MA, 1990.

[7] Giancarlo Aanna Maria Paola Bonacina. Kblab: an equational theorem prover for the macintosh. *CADE*, pages 548–550, 1989.

[8] Ursula Martin and Tobias Nipkow. Ordered rewriting and confluence. *CADE*, 10:365–380, 1990.

[9] David McAllester. Grammar rewriting. *CADE*, 11:124–137, 1992.

[10] Greg Nelson and Derec Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–395, 1980.

[11] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *JACM*, 2:191–216, 1986.

[12] Gerald E. Peterson. Complete sets of reductions with constraints. *CADE*, 10:381–395, 1990.