# Associative-Commutative Rewriting on Large Terms

Steven Eker

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA
Email: eker@csl.sri.com

**Abstract.** We introduce a novel representation for associative-commutative (AC) terms which, for certain important classes of rewrite rules, allows both the AC matching and the AC renormalization steps to be accomplished using time and space that is logarithmic in the size of the flattened AC argument lists involved. This novel representation can be cumbersome for other, more general algorithms and manipulations. Hence, we describe machine efficient techniques for converting to and from a more conventional representation together with a heuristic for deciding at runtime when to convert a term to the new representation. We sketch how our approach can be generalized to order-sorted AC rewriting and to other equational theories. We also present some experimental results using the Maude 2 interpreter.

## 1 Introduction

Rewriting modulo associative and commutative theories, otherwise known as AC rewriting, is a key operation in a number of languages [8, 4, 5, 14] and theorem provers [12, 21] employing term rewriting. In general, AC congruence classes are too large to be represented explicitly and so unique representatives, called *AC normal forms* or *AC canonical forms* are used. These are typically generated by replacing nested occurrences of the same AC operator by a *flattened* argument list under a variadic symbol, sorting these arguments under some linear ordering and combining equal arguments using multiplicity superscripts. For example the congruence class containing

$$f(f(\alpha, f(\beta, \alpha)), f(f(\gamma, \beta), \beta))$$

where $f$ is an AC symbol and subterms $\alpha$, $\beta$ and $\gamma$ belong to alien theories might represented by

$$f^*(\alpha^2, \beta^3, \gamma)$$

where $f^*$ is a variadic symbol that replaces nested occurrences of $f$. A more formal account of this transformation is given in [9]. Of course the linear ordering on terms used to sort the argument list must be extended to these AC normal forms. The multiset ordering has pleasant theoretical properties and is often used for this purpose, though as we will see later, it is seldom the best ordering for a practical implementation.

AC rewriting proceeds by using a special form of matching called *AC matching* on these AC normal forms (with some handling of extension, should an AC operator occur on top of the subterm being rewritten). Following the replacement step, the resulting term is generally not in AC normal form and must be renormalized before another AC rewrite step can be performed.

The problem of deciding whether an AC match exists is known to be NP-complete [3] and remains NP-complete even in the elementary case [11] where there is only a single AC operator with variables and constants. Nevertheless, instances that occur in practice tend not to be pathological and a variety of practical matching algorithms have been designed. There are really two cases that algorithms are designed for.

The first is the general case where arbitrary nonlinearity is allowed. This case tends to arise in theorem proving applications. Algorithms are generally search based, and use several techniques to collapse the search space, such as constraint propagation on nonlinear variables [15], recursive decomposition into subproblems via bipartite graphs [9], ordering matching subproblems based on constraint propagation considerations [10] and Diophantine techniques [11].

The second case is where there is little or no nonlinearity and the pattern falls into one of several forms for which efficient algorithms can be designed. This situation tends to arise when programming with AC rewrite rules. Examples are the depth-bounded patterns in the many-to-one matching algorithm used by ELAN [22, 20] and the greedy matching techniques adopted in Maude. Here a full AC matching algorithm of the first kind is needed as a fallback for when the specialized approaches fail to apply.

In recent years these has been much progress in improving the performance of AC rewriting implementations as evidenced by the CafeOBJ/Brute [17], ELAN [4] and Maude [5] systems. Nevertheless, even in the best case, all published algorithms for AC matching (supposing a successful match) and AC renormalization require the examination of all of the arguments in the flattened AC argument lists involved. Thus processing the elements of a set or multiset using AC rewriting will require at least quadratic time as each rewrite step will need at least linear time. This fact is a major hurdle in achieving the goal stated in [20] of promoting AC rewriting to a general purpose programming construct.

In this paper we introduce a new representation for terms in AC normal form and identify a broad class of AC patterns for which a single matching substitution can be computed in time logarithmic in the size of the AC argument list. Furthermore we identify an important class of righthand side replacement terms for which the AC renormalization step can also be accomplished in time logarithmic in the size of the AC argument list. Thus with some care about reduction strategy and the maintenance of *reduced flags* there is a very useful class of AC rewrite rules that can be executed in logarithmic time. We also examine a number of refinements to improve the performance of the basic method and sketch how it can be generalized to handle order-sorted rewriting. We also sketch how our general approach can be adapted to other equational theories. Because our novel AC representation can be cumbersome for more general algorithms and manipulations we discuss machine efficient techniques for converting between representations. To demonstrate the practicality of our approach we give the results of some initial experiments performed using various versions of the Maude 2 interpreter.

### 1.1 A Motivating Example

Maps, or associative arrays as they are sometimes called in programming, are a very useful data structure that generalize arrays. Consider the Maude 2 specification of the map data type shown in Figure 1. A more elegant specification could be given using an order-sorted signature and an identity but for clarity of exposition we will confine our discussion to the most basic language features.

The data type `Map` is formed from three constructors, one of which is declared to be AC. There are two defined operations, one of which creates a new map by inserting a new domain-range pair into an existing map and the other which looks for a domain-range pair having a given domain value in an existing map and returns the range value or the constant `undefined` if no such pair exists. Note that the last defining equation for each of these operations has the `owise` attribute, indicating that the equation should only be used if no regular equation applies.

Consider the equations defining the insert operation. The first two are for the basis cases, contain only free function symbols and could reasonably be expected to execute in constant time on any modern implementation of term rewriting. The same cannot be said for the third equation. In the case that a match is found, the substitution for $M$ will consist of all but one of the domain-range pairs underneath the variadic AC symbol. Using a traditional representation of AC argument lists, construction of this substitution will take linear time. A similar situation occurs with the second defining equation for the look-up operation. This contrasts with the $O(\log(n))$ time insert and look-up operations of the `map` class template in the C++ Standard Template Library or the `TreeMap` class in Java.

## 2 Basic Approach

We just consider the simple case of matching where we have a pattern term $p$ containing variables and a subject term $s$ which is ground. We wish to decide if there is a *matching substitution* $\sigma$ such that $p\sigma = s$, and if there is, compute a single such $\sigma$. We do not consider the generation of multiple substitutions, as would be needed to handle conditional equations.

We make the assumption that the same pattern $p$ will be used many times and so we allow arbitrary preprocessing and analysis to be done on $p$ before using it for matching.

```
fmod MAP is
  sorts Domain Range Map .
  op undefined : -> Range .
  op empty : -> Map .
  op _|->_ : Domain Range -> Map .
  op _,_ : Map Map -> Map [assoc comm] .
  var D : Domain .  vars R R' : Range .  var M : Map .

*** insertion
  op insert : Domain Range Map -> Map .
  eq insert(D, R, empty) = (D |-> R) .
  eq insert(D, R, D |-> R') = (D |-> R) .
  eq insert(D, R, (M, D |-> R')) = (M, D |-> R) .
  eq insert(D, R, M) = (M, D |-> R) [owise].

*** look-up
  op _[_] : Map Domain -> Range .
  eq (D |-> R)[D] = R .
  eq (M, D |-> R)[D] = R .
  eq M[D] = undefined [owise] .
endfm
```

**Fig. 1.** Maude 2 specification for the map data type.

Matching is inherently recursive and is performed by recursive decomposition into matching subproblems between the subterms of $p$ and the subterms of $s$. Solving matching subproblems where the subterms are headed by the same free function symbol is straightforward. We focus on solving matching subproblems where the subterms are headed by the same AC function symbol $f^*$.

### 2.1 Returning "undecided"

The idea is that our fast algorithm will solve the easy problem instances efficiently while leaving harder problem instances to a slower general purpose matching algorithm. Ideally we would be able to make the decision which algorithm to use before runtime, simply from an analysis of the pattern.

Unfortunately, if we adopt this approach, the class of patterns we can handle using the fast algorithm will be quite restricted. Our experience with the greedy matching algorithms used in the Maude 1 interpreter is that it is beneficial to handle a larger class of patterns with a fast algorithm, where the fast algorithm has the option of returning "undecided" if the matching problem turns out to be too hard (generally because of bindings to nonlinear variables that were unknown at analysis time) when the matching is attempted. When this happens, the time spent in the fast matching algorithm is wasted, and the slower general purpose algorithm must be run.

Of course we must be careful in selecting the class of patterns we will handle in the fast algorithm, to ensure that the "undecided" case is atypical, and that the time wasted in the fast algorithm when it does occur is small compared to the cost of running the slow algorithm.

### 2.2 Stripper-Collector Matching

We now describe our main algorithm, which we refer to as *stripper-collector matching*. Consider an AC matching subproblem

$$f^*(p_1^{k_1}, \ldots, p_n^{k_n}) \leq^?_{AC} f^*(s_1^{q_1}, \ldots, s_m^{q_m})$$

where subject subterms $s_1, \ldots, s_m$ are ground. Since this is a subproblem we assume that matching may have taken place above and that some set of variables $B$ has already been bound, yielding a partial substitution $\phi$. The goal of AC matching is to divide up the subject subterms $s_j$ among the pattern subterms $p_i$ and compute a substitution $\sigma$ consistent with $\phi$ such that if $p_i$ is assigned a single subject subterm $s_j$ then $p_i\sigma = s_j$ and if $p_i$ is assigned multiple subject subterms $s_{u_1}^{r_1}, \ldots, s_{u_k}^{r_k}$ then $p_i\sigma = f^*(s_{u_1}^{r_1}, \ldots, s_{u_k}^{r_k})$.

Our key idea is to represent the subject argument list $s_1^{q_1}, \ldots, s_m^{q_m}$ using an efficient set data structure. Then we restrict the subpatterns $p_i$ that we will handle in such a way that we can compute a solution with $p_1, \ldots, p_{n-1}$ getting one subject each, while everything else is given to $p_n$. We refer to $p_1, \ldots, p_{n-1}$ as *strippers* since they have the effect of stripping single arguments from the argument list while we refer to $p_n$ as the *collector* since is collects all the leftover arguments.

Finding the assignments for each of the strippers, $p_1, \ldots, p_{n-1}$, is done using $O(\log(m))$ time searches of the set data structure while the assignment to the collector, $p_n$ is computed using $(n-1)$ $O(\log(m))$ deletes from the set data structure. It is important that we choose assignments to strippers in such a way that should we fail at some stripper we know that there is no match and we do not have to backtrack and consider different assignments for earlier strippers. This leads to further restrictions on the subterms allowed for strippers and the order in which they must be handled. Pattern subterms with $n > 2$ tend to be fairly rare in programming applications so the in the most common case we will have $n = 2$ and we will need to do a single search followed by a single delete operation to compute a matching substitution.

In practice, terms will be represented by directed acyclic graphs, otherwise rewrites with nonlinear righthand sides become unreasonably expensive since variable bindings would need to be copied rather than shared. Thus we cannot do destructive updates on our set data structures since in general they will be shared with other subterms. What we need in fact, is a data structure for *persistent dynamic sets.*

### 2.3 Persistent Dynamic Sets

Dynamic sets are usually implemented using one of several balanced binary tree data structures [2, 1, 24]. Insertion and deletion operations can be made persistent by not storing back pointers, allocating new nodes rather than modifying existing nodes and rebuilding the path from a modification back up to the root.

For simplicity we use red-black trees with the particularly elegant persistent insertion algorithm due to Okasaki [23]. Deletion is a rather intricate operation which closely follows the regular red-black tree deletion algorithm given in [7, Chapter 14]. One difference is that, because we are creating new nodes rather than manipulating existing nodes, it pays to combine multiple rotations and avoid creating nodes that are then immediately discarded by the next rotation. Expanding out the four rebalancing cases given in [7, Chapter 14] we arrive at six cases together with their left-right reflections.

### 2.4 Augmenting the Data Structure

We use the terms in the AC argument list as the keys in our red-black trees. In each node we also maintain the multiplicity of the argument, along with another data item called *maximum multiplicity*. For a given node $N$, this value is the maximum multiplicity of any node in the subtree rooted at $N$ and can be computed incrementally in constant time. For a given multiplicity $k$, it allows us to decide in constant time whether a node with multiplicity greater or equal to $k$ exists, and if it does, to locate such a node in logarithmic time.

Insertion and deletion operations take a term and a multiplicity. In the case of an insertion of a term that already exists, rather than do a red-black tree insertion, we create a new node with an increased multiplicity and rebuild the path to the root. In the case of a deletion which does not remove all the multiplicity of the victim, rather than do a red-black tree deletion, we create a new node with a decreased multiplicity and rebuild the path to the root. Thus we are really simulating a multiset.

### 2.5 Partial Comparison

The existing linear ordering on terms is fine for efficiently locating a given ground term in a red-black tree, and the maximum multiplicity field allows us to efficiently find a ground term with a multiplicity greater or equal to a given value. For efficiently locating potential matches for nonground terms we need another tool. For simplicity we assume that all symbols are either AC or free. A *partial comparison function pc* takes a partial substitution $\phi$, a nonvariable, nonground term $t$ and a ground term $s$ and returns a result from the four element set $\{eq, lt, gt, \uparrow\}$ such that the following the properties hold.

- $pc(\phi, t, s) = eq$ implies that for every substitution $\sigma$ consistent with $\phi$, $t\sigma = s$.
- $pc(\phi, t, s) = gt$ implies that for every substitution $\sigma$ consistent with $\phi$, $t\sigma > s$.
- $pc(\phi, t, s) = lt$ implies that for every substitution $\sigma$ consistent with $\phi$, $t\sigma < s$.

Note that a partial comparison function is allowed to return $\uparrow$ everywhere.

A partial comparison function $pc$ is *reasonable* if whenever the top symbol of $t$ differs from that of $s$, $pc(\phi, t, s)$ is either $gt$ or $lt$. If $t$ and $s$ share the same free top symbol, and we use lexicographic ordering on arguments to linearly order terms with free symbols on top, we can do better. For example when $\phi$

contains assignments for variables $X_1, \ldots, X_n$, and $g$ is a free function symbol $pc(\phi, g(X_1, \ldots, X_n), g(s_1, , \ldots, s_m))$ should never need to return $\uparrow$.

Let $pc$ be a partial comparison function. Let $t$ be a nonvariable, nonground term and let $B$ be a set of variables. The pair $(t, B)$ is said to be *pc-complete* iff for every partial substitution $\phi$ that defines exactly the variables in $B$ and every ground term $s$, $pc(\phi, t, s) = \uparrow$ implies that there is a substitution $\sigma$ compatible with $\phi$ such that $t\sigma = s$. For example if $pc$ is reasonable, $t = f^*(X, Y)$ and $\{X, Y\} \cap B = \emptyset$ then the pair $(t, B)$ is $pc$-complete since $t$ will match any term with an $f^*$ on top.

Constructing a good partial comparison function depends on the term ordering chosen. From now on we will assume the existence of a reasonable partial comparison function $pc$.

### 2.6 Allowable Subpatterns

We now consider what class of subpatterns can actually be handled in the manner described above. For the collector we require that it be a linear variable; i.e. it must have multiplicity 1 and may not occur elsewhere in the term. For a stripper, there are a number of possibilities.

**ground subterm** Here we allow any multiplicity. Since a ground term must match an identical term in the argument list we can just search for that term in the red-black tree, and do a deletion operation. Of course if the term does not occur in the subject argument list with sufficient multiplicity we can return *failure*.

**bound variable** Here we also allow any multiplicity. During the analysis phase, when we decide on the order for matching subterms, we will know for each matching subproblem what nonlinear variables will already be bound. If the variable is bound to a term headed by a symbol other than $f^*$ we can treat it as a ground subterm. Otherwise we return "undecided".

**nonvariable, nonground subterm** Here we limit multiplicity to 1. Any variables occurring in the nonground term $p_i$ must either be bound or must not occur outside of $p_i$. Furthermore if there are multiple nonground strippers $p_{v_1}, \ldots, p_{v_k}$ they must have the property that for $c < d$, either $p_{v_c}$ is not unifiable with $p_{v_d}$ (no possible conflict over subject subterms) or $p_{v_d}$ matches $p_{v_c}$ ($p_{v_d}$ is at least as general as $p_{v_c}$). This restriction ensures that an earlier stripper will not take a subject subterm needed by a later one. We can find a first candidate subject term for assignment to $p_i$ by using our reasonable partial comparison function $pc$ to find the leftmost subject term $s_j$ such that $pc(\phi, p_i, s_j)$ is *eq* or $\uparrow$. If we want to ensure that $p_i$ matches $s_j$ we need also enforce the restriction that $(p_i, B)$ is $pc$-complete. Since this is quite strict, a practical alternative is to just use $s_j$ as a starting point for a match for $p_i$, and keep searching, left to right, until we get a match or we reach an $s_k$ such that $pc(\phi, p_i, s_j) = lt$ in which case we can return fail. Of course once we start searching we require linear time in the worst case. Another alternative is to return "undecided" if $p_i$ fails to match $s_j$. A stripper of this form must be handled after strippers of the first two forms to avoid "using up" a subterm needed by them.

**high multiplicity unbound variable** We allow at most one of these and only if there are no nonvariable, nonground strippers. Also the variable must not occur anywhere else in the term (i.e. it is *almost linear* in the sense of [20]). We can find a subject subterm with adequate multiplicity to assign to it in $O(\log(m))$ time if such a subterm exists by following the maximum multiplicity fields in the red-black tree. A stripper of this form must be handled after strippers of the first two forms to avoid "using up" a subterm needed by them.

**linear variable** Since linear variables can take anything we can just assign to them anything leftover after the other strippers have been dealt with. This form of stripper is alway handled last.

### 2.7 A Concrete Example

Consider the matching problem

$$h(f^*(g(D,R),M),D) \leq^?_{AC} h(f^*(g(1,1),g(2,4),\ldots,g(100,10000)),42)$$

where $h$ and $g$ are free binary function symbols and $D$, $R$ and $M$ are variables. Matching the subterms under the free function symbol $h$ results in a partial substitution $\phi$ such that $\phi(D) = 42$, and a matching subproblem

$$f^*(g(D,R),M) \leq^?_{AC} f^*(g(1,1),g(2,4),\ldots,g(100,10000)) \ .$$

Now assume we use lexicographic ordering on arguments to linearly order terms with free function symbols on top, and the usual ordering on integer constants. We can design a partial comparison function $pc$ such that $pc(\phi, g(D,R), g(i,j))$ returns $\uparrow$ when $i = 42$ and either $lt$ or $gt$ otherwise. Furthermore, $(g(D,R),\{D\})$ will be $pc$-complete. Thus we can use $pc$ to efficiently locate a subject subterm to match against $g(D,R)$ if one exists.

### 2.8 Comparison with Many-to-One AC Matching

There are similarities between the class of AC patterns we handle and those handled by the many-to-one AC matching used in ELAN [22, 20]. There are two key differences however.

1. We have the restrictions on nonvariable, nonground terms and variables with high multiplicity in order to avoid the need for backtracking.
2. We handle nonlinear variables, as long as they have already been bound by the time the AC matching takes place.

One possible weakness of many-to-one AC matching with large AC terms can be illustrated by the following argument. Suppose we have $k$ patterns each with $n$ subterms under an AC symbol and we have a subject with $m$ subterms under an AC symbol. We consider the worst case: there are no common subpatterns so the many-to-one AC matching algorithm must consider $k.n.m$ combinations of pattern and subject subterms; also, all but the last pattern fails to match so any sequential algorithm must attempt all $k$ patterns. If the sequential algorithm can detect failure quickly for some of the failing patterns it can avoid considering many of the $k.n.m$ combinations of pattern and subject subterms. In the Maude 1 interpreter, some of the special case greedy AC matching algorithms can detect failure in $O(log(m))$ time via binary search even though computing a matching substitution takes linear time. This appears to be the main reason why the Maude 1 interpreter can hold its own against the ELAN compiler, at least for for unconditional AC rewriting [20]. Failing fast is important! The key advantage of our new algorithm over that used in Maude 1 is that not only can we often detect failure in $O(log(m))$ time, but also, in the success case, we can often compute a matching substitution in $O(log(m))$ time.

### 2.9 AC Renormalization

Following a successful match and replacement, we need to convert the resulting term back into AC normal form. We assume that subterms are already in AC normal form, either because of bottom-up renormalization or because they were assignments to variables. The case we can deal with efficiently is of the form

$$f^*(t_1^{k_1}, \ldots, t_n^{k_n})$$

where for some $i$, $t_i$ is an AC normal form with top symbol $f^*$, $k_i = 1$ and for $j \neq i$, $t_j$ has a top symbol different from $f^*$. In this case we can compute the AC normal form by doing $n - 1$ insertions into the argument list of $t_i$.

A useful heuristic for deciding when to convert to red-black tree representation in the first place is to set some threshold, $T$. When we arrive at a renormalization situation of the above form, but $t_i$ currently has its argument list stored in a more conventional representation such as a vector or list, we convert the argument list to a red-black tree if the length of this vector or list exceeds $T$.

## 3 Refinements and Generalizations

We now sketch some refinements and generalizations of our basic approach.

### 3.1 Choice of Ordering on AC Normal Forms

Linear term orderings typically start with some ordering on function symbols and are then built up inductively. In the free theory, lexicographic ordering on arguments is often used. When we come to compare a pair of AC normal forms

$$\alpha = f^*(\alpha_1^{k_1}, \ldots, \alpha_n^{k_n}) \text{ and } \beta = f^*(\beta_1^{j_1}, \ldots, \beta_m^{j_m})$$

with the same top function symbol we need to choose some linear ordering on their argument lists. Although the choice of ordering will in general affect the order in which matching substitutions are found and can therefore affect the efficiency of reduction, it seems hard to predict what effect a given ordering will have on an arbitrary AC term rewriting system. Instead we seek a linear ordering that can be computed quickly in the average case. Typically multiplicity comparisons require several machine instructions while term comparisons involve a recursive call. Thus we would like to minimize the number of subterm comparisons when $\alpha \neq \beta$. One technique is to first compare $n$ and $m$. If they are equal we then do lexicographic comparison on the sequences of multiplicities, $k_1, \ldots, k_n$ and $j_1, \ldots, j_m$. Only if these sequences are equal do we finally do a lexicographic comparison on the sequences of subterms themselves. One drawback of of this method is that it requires two passes over the argument lists and in the common case that all the multiplicities are 1 the first pass is a waste of time. An alternative in the case that $n = m$ is to do a single pass lexicographic comparison on the sequences $k_1, \alpha_1, \ldots, k_n, \alpha_n$ and $j_1, \beta_1, \ldots, j_m, \beta_m$ of interleaved multiplicities and subterms. This is the ordering adopted in Maude.

### 3.2 Order-Sorted Rewriting

Order-sorted rewriting is a convenient extension of many sorted rewriting [13]. It complicates our basic algorithm in two ways. Firstly we need to maintain a sort for each reduced subterm. Secondly matching can fail because of sort considerations. Let $s_{f^*} : \mathit{Sort} \times \mathit{Sort} \to \mathit{Sort}$ be the sort function for $f^*$; it is necessarily associative and commutative. We avoid linear time sort computations by maintaining a sort in each red-black tree node. By adding an artificial identity to $s_{f^*}$ we can calculate the sort of $f^*(t^k)$ in $O(\log(k))$ time using the classical monoid powering by squaring algorithm. When matching fails because of sort considerations, we have to return "undecided"; we minimize the risk of this by always choosing the linear variable of largest sort as the collector.

### 3.3 Other Equational Theories

Our basic approach of treating AC-matching as one or more deletions and AC renormalization as one or more insertions in a suitable persistent data structure can be generalized to other equational theories. We briefly sketch a couple of important cases.

Suppose we add an identity to an AC function symbol to get an ACU function symbol (the U stands for *Unit*). ACU normalization is very similar to AC normalization except that we do not include identity elements; this means that before doing an insert operation into an ACU argument list we check if the element to be inserted is the identity and do not insert it in that case. ACU matching is complicated by the possibility of collapses that bring alien nonground arguments into their parents theory. But these situations are atypical and can be detected by pattern analysis, in which case a slow general purpose matching algorithm can be used. Collapses also complicate the definition of partial comparison. The other distinctive feature of ACU matching is that variables under an ACU symbol might be assigned the identity element; possibly causing the collapse in the ACU pattern itself. This is just another case to be checked for in the fast algorithm where the matching process would otherwise fail.

A more interesting situation arises when we remove the commutative axiom to get a purely associative function symbol. The case has practical importance because it corresponds to lists. We can get an associative normal form by flattening nested function symbol occurrences without sorting the arguments. A fast matching associative algorithm proceeds by stripping arguments from one or both ends of the subjects associative argument list. Until we encounter a variable subpattern, the correspondence between subpatterns and subject arguments is forced, and there is no restriction on nonvariable subpatterns in this case.

An appropriate data structure for representing associative argument lists is the *Persistent Catenable Deque* of Kaplan and Tarjan [19] or its amortized time simplification [18]. This remarkable if highly intricate data structure supports persistent insertions and deletions from both ends of list together with persistent concatenation, all in constant time. The constant time persistent concatenation, in particular, allows unrestricted associative renormalization, and also allows the construction (by self concatenation) of terms whose explicit representation would be exponential.

Using this data structure to represent associative normal forms, a broad class of associative rewrite rules could be executed in constant time; i.e. with the same complexity as syntactic rewriting.

## 4 Practical Considerations

We now discuss some of techniques needed for a machine efficient implementation of our approach.

### 4.1 Iterators

The first thing we need is an efficient way of making an in-order traversal of the argument list under an AC operator without the inconvenience of using recursion. Because of the need for persistence, the nodes in our red-black trees cannot hold back pointers to their parents. Thus we need to hold the path from the root to the current node on an explicit stack. The number of possible red-black tree nodes is bounded by the address space of the machine and the height of the largest red-black tree is bounded by twice the logarithm of this number. Thus we can use small fixed size stacks for iterators. By using fixed size C/C++ arrays to hold these stacks and machine pointers to the top elements, each of the usual stack operations can be implemented in a couple of machine instructions.

An iterator is initialized by stacking the path to the leftmost node in the red-black tree. To find the next node following an in-order traversal we consider two cases:

1. If our current node has a right child we stack this right child together with the path to it's leftmost descendant; otherwise,

```
make_tree(vec, first, size, is_red)
    if size = 0 then
        return nil
    else
        left_size := ⌊size/2⌋;
        right_size := size − 1 − left_size;
        left_tree := make_tree(vec, first, left_size,
                                left_size > right_size and pow2min1(left_size));
        right_tree := make_tree(vec, first + left_size + 1, right_size, false);
        return new_node(vec[first + left_size], left_tree, right_tree, is_red)
    fi
```

**Fig. 2.** Algorithm for constructing a red-black tree from a vector.

2. we pop the stack until the top of the stack refers to a node whose right child is not the node we just popped; if we end up with an empty stack we have completed the traversal.

Thus our implementation becomes a pair of tight loops - one for descending leftmost paths and one for ascending rightmost paths.

For a number of applications, most notably converting a red-black tree to a sequence when we wish a more conventional term representation, it is not important to have the full path to the current node stored on the stack. In this situation we can optimize the traversal algorithm by unstacking the parent node when taking a right branch. Then in case 2 above, we need only pop a single element of the stack. This technique can be viewed as the explicit stack equivalent of the well known tail recursion elimination optimization.

### 4.2 Red-Black Tree Construction

In order to convert from a conventional term representation to our new representation we need an algorithm for constructing red-black trees from a given ordered sequence. A detailed analysis of this problem together with linear time solutions is given by Hinze [16]. However in the case that our sequence is actually stored as a vector we can use the simpler recursive algorithm shown in Figure 2. The idea is that, at each stage we chose the middle element, preferring the rightmost element in the case of a tie. We then create a node with this element and execute the algorithm recursively on the left and right subarrays to generate the left and right subtrees. It is easy to see that the resulting binary tree will be balanced; the subtle point is to generate node colors that preserve the red-black property. The idea is that if the left subtree is larger than the right subtree there will need to be red nodes in the left subtree to reduce the black height of at least some of its leaves. Furthermore, if the left subtree contains exactly $2^n - 1$ nodes for some $n$ (tested for by the *pow2min1* function) then we have a complete binary subtree and we can color its root red, reducing the black height of all of its leaves.

It may not be immediately obvious how to efficiently test if some positive integer $k$ is of the form $2^n - 1$, but in fact the *pow2min1* function can be implemented with 3 or 4 machine instructions using the following C/C++ bit twiddling trick: `((k + 1) & k) == 0`. We have presented the red-black tree construction algorithm in recursive form for clarity of exposition but for a real implementation it can be made nonrecursive using an explicit fixed size stack.

### 4.3 Memory Usage

A vector based representation of AC arguments lists typically uses two machine words per argument, to hold a pointer to the argument and its multiplicity. A linked list representation requires one or two additional machine words per argument depending on whether it is singly or doubly linked.

Our persistent red-black tree representation requires four additional machine words per argument over the vector based representation, to hold the maximum multiplicity, the left and right subtree pointers, and the node color. In the order-sorted generalization, the sort information can share a machine word with the color flag. However unlike the vector and linked list approaches where only subterms are shared, with the red-black tree representation, large parts of the argument lists can be shared between different AC subterms. Depending on the term rewriting system, this can more than compensate for the increase in the per argument cost of storing the argument lists.

## 5 Experimental Results

To demonstrate the feasibility of our approach we added a red-black tree implementation of AC argument lists and the associated matching and renormalization algorithms to the Maude 2 interpreter [6]. The Maude 2 rewrite engine is otherwise an incremental improvement over that in Maude 1, which already has very competitive AC rewriting performance [20]. The highly modular design of the rewrite engine makes it ideal for this kind of experiment since all rewriting theories plug into an abstract theory interface which provides generic term manipulation primitives. It is therefore easy to support multiple term representations and matching algorithms for a given theory with no effect on the rest of the system.

Consider the following recursively defined function on the natural numbers:

$$f(0) = 0$$

$$f(n+1) = f(\lfloor n/2 \rfloor) + f(\lfloor n/4 \rfloor)$$

This definition is somewhat reminiscent of the famous Fibonacci function. Indeed the only reason we do not use the Fibonacci function is that rapid growth in the size of the numbers involved obscures the performance of the non-numerical rewriting. In particular, we need two recursive calls and thus will end up using

```
fmod MAP-TEST is including MAP .  protecting NAT .
  subsort Nat < Domain Range .
  var N : Nat .  vars M M' : Map .
  op f : Nat Map -> Map .
  eq f(s N, M) = insert(s N,
                        ((f(N, M))[N quo 2]) + ((f(N, M))[N quo 4]),
                        f(N, M)) .
  eq f(0, M) = insert(0, 1, M) .
endfm

red f(100, empty)[50] .
red f(1000, empty)[500] .
red f(10000, empty)[5000] .
red f(100000, empty)[50000] .
red f(1000000, empty)[500000] .
```

**Fig. 3.** Benchmark problem for map specification.

| Size | Rewrites | Vector/legacy | | | Vector/SC | | | Red-black tree/SC | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | seconds | rw/sec | MBs | seconds | rw/sec | MBs | seconds | rw/sec | MBs |
| 100 | 703 | 0.02 | 35150 | 5.2 | 0.01 | 70300 | 5.7 | 0.01 | 70300 | 5.7 |
| 1000 | 7003 | 0.71 | 9863 | 6.0 | 0.32 | 21884 | 6.6 | 0.11 | 63663 | 6.6 |
| 10000 | 70003 | 158.82 | 440 | 20 | 71.74 | 975 | 20 | 1.61 | 43480 | 19 |
| 100000 | 700003 | 17496.59 | 40 | 187 | 8766.46 | 79 | 187 | 20.73 | 33767 | 138 |
| 1000000 | 7000003 | - | - | - | - | - | - | 306.48 | 22839 | 1373 |

**Table 1.** Performance in seconds, rewrites/second and megabytes used.

exponential time unless we use some kind of memoization or dynamic programming technique. While Maude has built-in support for memoization, we will keep an explicit memoization table using the map data type defined earlier in Figure 1, in order to test the speed of the AC rewriting. The Maude code is shown in Figure 3. The function computes a map rather than a single value, and in order to have a concise output, we simply evaluate it near the middle of the range. Note that the subexpression $f(N, M)$ occurs three times in the right hand side of the first equation, but it is only evaluated once because Maude combines common subexpressions.

The results for three versions of Maude 2 running on a 550MHz UltraSPARC IIi with 1.5GB of RAM are shown in Table 1. The first version (Vector/legacy) uses the matching algorithms and vector based AC argument list representation inherited from Maude 1, which includes a primitive version of the stripper-collector matching algorithm that does not use partial comparison. The second version (Vector/SC) uses the full stripper-collector matching algorithm presented here but retains the vector based AC argument list representation. The third version (Red-black tree/SC) uses the full red-black tree based stripper-collector matching and renormalization algorithms. The two vector based versions failed to terminate after a cpu-day while attempting the last reduction. Here the runtime grows slightly worse than quadratic while rewriting speed falls off slightly worse than linearly with the full stripper-collector version about twice as fast as the legacy version. With the red-black tree representation, the runtime growth is approximately $n \log(n)$ while rewriting speed falls off approximately logarithmically. All versions show a somewhat bigger than expected slowdown going from size 1000 to size 10000, probably due to L2 cache effects.

## 6 Concluding Remarks

Our method is somewhat intricate and much care is required in the implementation to avoid linear time algorithms for other parts of the execution process such as term traversal and sort computation. Nevertheless we have shown that our method can provide an approximately $n/\log(n)$ speed up over the already fast algorithms used in Maude 1 for large $AC$ terms and simple $AC$ rewrite rules. We plan to implement persistent data structure based rewriting algorithms for other equational theories in Maude 2 in the near future.

**Acknowledgements** This work was partly funded by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130. I thank Pat Lincoln, José Meseguer, Carolyn Talcott and the anonymous referees for their comments on earlier drafts.

## References

1. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.
2. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.
3. Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1–2):203–216, February 1987.
4. Peter Borovanský, Claude Kirchner, Helènė Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Journal of Theoretical Computer Science*, 285(2):155–185, August 2002.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Journal of Theoretical Computer Science*, 285(2):187–243, August 2002.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. This volume.
7. Thomas H. Cormen, Charles E. Leierson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.
8. Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. *Journal of Theoretical Computer Science*, 285(2):289–318, August 2002.
9. S. M. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

10. Steven Eker. Fast matching in combinations of regular equational theories. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.
11. Steven Eker. Single elementary associative-commutative matching. *Journal of Automated Reasoning*, 28(1):35–51, January 2002.
12. S. J. Garland and J. V. Guttag. An overview of LP, the Larch Prover. In N. Dershowitz, editor, *Proceedings of 3rd International Conference on Rewriting Techniques and Applications (RTA '89)*, number 355 in Lecture Notes in Computer Science, pages 137–151. Springer-Verlag, April 1989.
13. J. Goguen and J. Meseguer. Order sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, November 1992.
14. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ3. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, March 1992.
15. Bernhard Gramlich and Jörg Denzinger. Efficient AC-matching using constraint propagation. Technical report, FB Informatik, Universität Kaiserslautern, Pf. 3049, D-6750, Kaiserslautern, Germany, 1988.
16. Ralf Hinze. Constructing red-black trees. In Chris Okasaki, editor, *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, volume CUCS-023-99 of *Computer Science Technical Reports*, pages 89–99. Columbia University, September 1999.
17. M. Ishisone and T. Sawada. Brute: brute force rewriting engine. In *Proceedings of the CafeOBJ Symposium '98, Numazu, Japan.* CafeOBJ Project, April 1998.
18. Haim Kaplan, Chris Okasaki, and Robert E. Tarjan. Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977, 2000.
19. Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slowdown. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC '95)*, pages 93–102. ACM Press, May 1995.
20. Helèné Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, March 2001.
21. William McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, December 1997.
22. Pierre-Etienne Moreau and Helèné Kirchner. A compiler for rewrite programs in associative-commutative theories. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming (PLILP/ALP '98)*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998.
23. Chris Okasaki. Functional pearl: Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.
24. Salvador Roura. A new method for balancing binary search trees. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001)*, number 2076 in Lecture Notes in Computer Science, pages 469–480. Springer-Verlag, July 2001.