Associative-Commutative Rewriting on Large Terms

Steven Eker, SRI

AC Normal Forms

Congruence classes modulo the AC axioms for some set of function symbols are given a unique representative call the AC Normal Form or AC Canonical Form. Typically computed by:

1. Replacing nested occurrences of the same AC function symbol, f, by a single variadic function symbol f^*.
2. Sorting the flattened argument list according to some linear ordering.
3. Combining identical subterms using multiplicity superscripts.

For example:

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

where f is an AC symbol might represented by

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

AC Rewriting

AC congruence class rewriting is usually simulated by AC matching on AC normal forms and AC renormalization following replacement. Some handling of extension is needed for l.h.s. patterns with an AC symbol on top.

AC matching is NP-complete (due to nonlinear variables).

Two kinds of algorithms in practice:

1. Slow general purpose algorithms based on searching with various techniques to cut down the seach space.
2. Fast algorithms that handle common special cases, typically restricting the occurrences of nonlinear variables — Elan depth bounded patterns, Maude greedy matching.

Most programming applications of AC rewriting tend to use simple patterns.

The Problem

All published algorithms for both AC matching and AC renormalization examine every argument underneath an AC function symbol even in the best case.

This may be acceptable where the AC symbol is used as the operator in some commutative semigroup ($+, \times$ on numbers, $\vee, \wedge$ on Booleans etc.) if the size of the expressions do not grow to large.

If the AC symbol is used to represent a set, multiset or map, the cost of each operation becomes at least linear in the size of the data structure - unacceptable for general purpose programming.

Sets, multisets and maps usually have log-time operations in conventional programming languages/libraries.

Would like to have like to have efficient data types definable by AC rewriting.

A Motivating Example

Want to represent a map as a set of (domain, range) pairs; for example:

```
1 |-> 1, 2 |-> 4, 3 |-> 9, 4 |-> 16, 5 |-> 25
```

```
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 .
```

A Motivating Example (2)

```
*** 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
```

AC Matching

We consider matching in a single AC layer, where matching may have been performed above, producing a partial substitution ϕ.

AC matching essentially consists of dividing up the arguments in the subject amongst the arguments in the pattern and computing a substitution σ consistent with ϕ such that each pattern argument σ-matches the subject arguments allocated to it. For example:

$$\phi = \{X \mapsto c\}$$

$$f^*(g(X), g(Y), Z) \leq^?_{AC} f^*(g(a), g(b), g(c), g(d), g(e))$$

$$\sigma = \{X \mapsto c, \quad Y \mapsto a, \quad Z \mapsto f^*(g(b), g(d), g(e))\}$$

Stripper-Collector Matching

Consider an AC matching subproblem

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

with partial substitution ϕ.

Represent the subject argument list $s_1^{q_1}, \ldots, s_m^{q_m}$ using an efficient set data structure.

Compute a match where each of $p_1, \ldots, p_{n-1}$ (called strippers) get one subject argument and p_n (called the collector) gets everything else.

Finding a match for each stripper is done with a log-time search.

Computing the match for the collector is done with $n-1$ log-time deletes from the set data structure.

Persistent Dynamic Sets

Terms will typically be shared (e.g. the map look-up function will usually share the map subterm with its caller). Therefore we cannot change original argument list.

Dynamic sets are usually implemented using balanced binary trees.

Insertion and deletion can be made persistent by

- Not storing back-pointers (use an explicit stack);
- Allocating new nodes rather than modifying old nodes.
- Rebuilding the path from a modification back up to the root.

Augmenting the Data Structure

Each node N of the binary tree holds:

1. A pointer to the argument term (used as the key)
2. The multiplicity of that argument.
3. The maximum multiplicity of all nodes in the subtree rooted at N.

The maximum multiplicity field can be computed incrementally in constant time.

It allows us to determine in constant time if there is a node with multiplicity $\geq n$ and if so, locate one such node in log time.

Insertion and deletion operations take a term *and* a multiplicity - we are really simulating a multiset.

Partial Comparison

To locate potential matches for nonground subpatterns we need a partial comparison function, pc.

Let ϕ be a partial substitution, t be a nonground term, s be a ground term.

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

pc can also return $\uparrow$ for "don't know". A naive partial comparison function just looks at the top symbols of t and s. Smarter versions look at arguments under free function symbols and take account of variables bound in ϕ. For example, assuming g is free and we use the alphabetic ordering on constants we might have:

$$pc(\{X \mapsto b\}, g(X, Y), g(a, c)) = gt$$

Compromises

We assume the pattern size is a constant - reasonable for a fixed term rewriting program. Ideally we would like to handle each subpattern under an AC function symbol using time that is logarithmic in the size of the subject. In practice, to increase the applicability of the method, it is useful to make compromises:

1. We attempt more general subpatterns than we are certain to be able to handle; and we return undecided if it looks too hard when we see the subject or partial substitution.
2. We will resort to linear search in certain places to avoid returning undecided since fallback general algorithm will best case linear.
3. We will compare variable bindings to subject subterms even though the sizes of the terms involve are not fixed by the pattern, and term comparison could take linear time in pathological cases.

What Subpatterns Can We Handle?

Collector must be an unbound linear variable. Possibilities for strippers:

Ground subpatterns and bound variables are easy - just locate the term in the binary tree and check that it has sufficient multiplicity. We handle these first.

Unbound linear variables are also easy since they can match anything. We handle these last.

Unbound nonlinear variables are harder; we only handle these if they don't occur elsewhere in term. We sort them in order of decreasing multiplicity. We locate matches using maximum multiplicity field. If more than two of these then we must return undecided if we fail:

$$f^*(W^3, X^2, Y^2, Z) \leq^?_{AC} f^*(a^4, b^3, c)$$

What Subpatterns Can We Handle? (2)

Nonvariable, nonground subterms are hardest. For a nonvariable, nonground subterm p, we limit its multiplicity to 1 and require that any unbound variables occurring in p do not occur outside of p.

If there are multiple nonvariable, nonground subterms then they must be able to be ordered such that if q is after p then either q is not unifiable with p (no conflict over subject arguments) or q is more general than p.

We do not allow both unbound linear variables and nonvariable, nonground subterms.

For each p, we find a leftmost potential subject argument s for each p such that $pc(\phi, p, s) = eq$ or $\uparrow$. if this fails we default to a linear left-to-right search until we reach a subject s' such that $pc(\phi, p, s') = lt$ and we can return failure.

AC Renormalization

Subterms under an AC function symbol are assumed to be in AC normal form either because of bottom-up renormalization or because they were assignments to variables.

We can handle terms of the form

$$f^*(t_1^{k_1}, \ldots, t_{m-1}^{k_{m-1}}, f^*(\ldots), t_{m-1}^{k_{m-1}}, \ldots, t_n^{k_n})$$

where $t_i^{k_i}$ is alien by inserting the aliens into the binary tree of the inner f^* term with $n-1$ log time inserts. We assume n is constant for a given term rewriting program.

We convert from vector to binary tree representation of argument lists when we reach the above situation, and the inner f^* term has its arguments in vector form, and they exceed some threshold.

Conversion back to vector form is done by necessity when fast matching/renormalization algorithms don't apply.

Generalizations

The idea of matching as deletion and renormalization as insertion in a suitable persistent data structure carries over to other situations.

Order Sorted Rewriting: Avoid linear time sort computations by keeping sort information in each node of the binary tree and using monoid powering algorithm to deal with multiplicities More ways to fail; choose linear variable of largest sort as collector.

AC + Unit: Renormalization avoids inserting identity elements. Pathological collapses can be detected and excluded at analysis time. Variables may be assigned identity.

Associative Rewriting: Flatten nested function symbols to get a list of arguments under a variadic function symbol. Patterns generally strip arguments from either end. Renormalization requires inserts and/or concatenation. Kaplan-Tarjan *Persistent Catenable Deque* allows inserts, deletes at either and concatenation in constant time.

Practical Condiderations

Explicit stacks for binary tree manipulation will have small bounded size ($< 2\times$ pointer length for red-black trees). Fixed size arrays can be allocated on the system stack for efficiency.

Traversing a binary tree using an explicit stack requires constant amortized time per node with a very low constant factor.

Constant factor can be made even lower by explicit stack equivalent of tail end recursion elimination if full path from current node to root not required.

Red-black trees can be constructed from arrays in linear time with a very low constant factor and no wasted node allocations.

Memory requirement is four machine words (left and right pointers, maximum multiplicity, color) per argument more than array representation but parts of argument lists are usually shared.

Experimental Results

Use map data structure data structure for memoization when evaluating recursive function with two recursive calls in the r.h.s.

```
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(1000000, empty)[500000] .
```

Experimental Results (2)

Three versions of Maude 2 running on a 550MHz UltraSPARC IIi with 1.5GB of RAM.

Vector/legacy:	Vector representation, Maude 1 stripper-collector algorithm.
Vector/SC:	Vector representation, Full stripper-collector.
Red-black tree/SC:	Red-black tree representation, Full stripper-collector.

Size	Rewrites	Vector/legacy		Vector/SC		Red-black tree/SC	
		seconds	rw/sec	seconds	rw/sec	seconds	rw/sec
100	703	0.02	35150	0.01	70300	0.01	70300
1000	7003	0.71	9863	0.32	21884	0.11	63663
10000	70003	158.82	440	71.74	975	1.50	46668
100000	700003	17496.59	40	8766.46	79	19.32	36232
1000000	7000003	-	-	-	-	274.42	25508

Roughly an $n/\log(n)$ speed up.

Experimental Results (3)

Using Hsiang's term rewriting system for propositional calculus to check random tautologies, sorted by number of rewrites required.

Maude 1.0.5			Maude 2.0			Speed up
rewrites	seconds	rw/sec	rewrites	seconds	rw/sec	
554,819	8.65	64140	554,823	5.590	99252	1.55
1,936,873	33.77	57354	1,936,879	19.36	100045	1.74
3,381,995	95.03	35588	3,381,985	37.68	89755	2.52
4,772,771	231.43	20622	4,772,769	60.40	79019	3.83
7,114,821	173.02	41121	7,114,827	82.81	85917	2.09
9,034,912	697.68	12949	9,034,912	113.53	79581	6.15
12,771,549	882.84	14466	12,771,539	162.36	78661	5.44
17,224,917	1241.30	13876	17,224,919	211.50	81441	5.87
24,271,013	2246.17	10805	24,271,011	311.09	78019	7.22
36,615,000	2781.72	13162	36,615,006	467.48	78324	5.95

Future Work

Better criteria for switching from conventional to persistent representation. Perhaps keep a history for each function symbol to decide if it benefits from the persistent representation of its arguments.

Better partial comparison functions (be able to carry partial comparison through theories other than the free theory).

Hashing techniques to speed up term comparisons.

Handle non-eager evaluation strategies, possibly by adding extra flags to the binary tree nodes to avoid unnecessary re-traversals.

Compiling AC rewriting systems to C - should be straightforward.

Future Work (2)

Generate multiple matches for handling conditional equations and rewrite search search:

1. May not want all solutions - for example we don't care about solutions that differ only on variables that don't occur in the condition.
2. Want to generate new matches on demand, since we may find an acceptable one early on.
3. Want to share work between matches.

Tentative support for this in the Maude 2.0 release.