The Design of an Extensible High Performance Term Rewriting Engine

Steven Eker

Computer Science Laboratory
SRI International
Menlo Park, CA

October 27, 2015

Implementing Term Rewriting

- Different applications and issues; different answers depending on various assumptions.
- Reduction strategy: lazy vs. eager, sequential vs. parallel, executions vs. search.
- Restrictions on term rewriting system: left linear, orthogonal, confluent, terminating, constructor discipline.
- Extensions: conditional rewriting, rule precedences, congruence class rewriting, user definable strategies, strictness annotations, order-sorted rewriting, higher-order.

Term Rewriting Basic Tasks

- (Theory normalization: find unique representative of congruence class.)
- Redex location: select a rule and locate a redex according to some strategy; compute a substitution.
- Replacement: Construct and instance of rules righthand side using substitution.
- (Sort calculation: compute sort of node in the term graph.)
- Garbage collection: reclaim unreachable nodes.

Transparency 1

Advantages of actually doing the rewrites:

- Can offer tracing (real-time or as a rewrite proof).
- Can show the user the current state after a ctrl-C interrupt.
- Can offer single stepping, break-points.

Naively performing the rewrites means we always retain an internal state that coincides with the user's model of what is going on — the interpreter look and feel.

Transparency 2

Alternative is to keep pending evaluations on the stack and only build term-graph nodes when they are reduced up to strategy.

- Save time building nodes (stacking is usually cheaper than consing).
- Nice implementation as C function calls (using the C stack).
- Hash consing feasible (constructed nodes aren't rewritten).
- Generational garbage collection feasible (no pointers from old to new).
- No need for in-place replacement (with the problem that replaced node may be smaller than replacement).

If we are never going to let the user see the internal workings we could also try transforming the user's term rewriting system.

Free Theory Matching (naive)

The obvious recursive algorithm does a depth-first traversal of pattern and subject:

1. If pattern and subject have the same function symbol, match on their corresponding subterms.
2. If pattern and subject have different function symbols, fail.
3. If pattern has an unbound variable, bind it to subject.
4. If pattern has a bound variable, compare binding to subject and fail if not equal.

Most failures occur due to (2) — time wasted binding variables.

Free Theory Matching (one-to-one) 1

Faster alternative compiles pattern into a (non-branching) automaton.

$$f(g(X, Y), h(X, Z))$$

Λ	test for f
$\Lambda.1$	test for g
$\Lambda.2$	test for h
$\Lambda.1.1$	bind X
$\Lambda.1.2$	bind Y
$\Lambda.2.2$	bind Z
$\Lambda.2.1$	compare X

Free Theory Matching (one-to-one) 2

- Save term pointers in array locations (semi-compilation) or registers (full compilation) determined at compile time for constant time access to positions in subject.
- Three tight loops (semi-compilation) or linear code (full compilation).
- Match symbols before binding variables.
- Bind all variables before checking nonlinear variables.

Free Theory Matching (many-to-one) 1

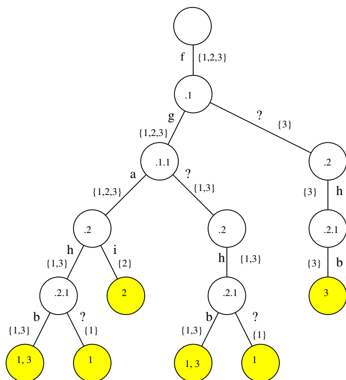
- Even faster alternative compiles (possibly linearly ordered) set of patterns into an automaton (aka discrimination net, decision tree, decision diagram).
- Consider patterns

$$f(g(X, Y), h(X, Z)) \tag{1}$$

$$f(g(a, X), i(Y)) \tag{2}$$

$$f(X, h(b, Y)) \tag{3}$$

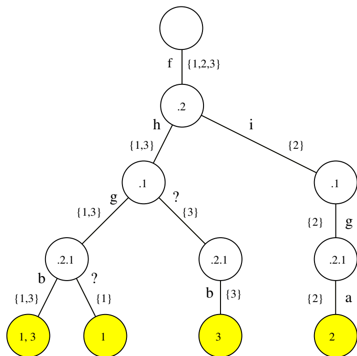
Free Theory Matching (many-to-one) 2



Free Theory Matching (adaptive traversal) 1

- No need to traverse subject depth-first left to right.
- Adaptive traversal can produce automata with shorter average path length, short maximal path length, fewer states etc.
- Finding optimal automaton under most reasonable criteria is known to be NP-hard.
- Various heuristics are effective; for example choose the position that minimizes the sum of the number of live patterns over all exiting arcs.
- Automaton can be compiled to nested switch statements for a compiled implementation.

Free Theory Matching (adaptive traversal) 2



Free Theory Matching (Maude implementation)

- Need to find all matching rules (because rule may fail due to sorts of variables being to small, clash on non-linear variable, failure of condition)
- Interpretation: Ternary tree in order to use a single tight loop; index by sign bit trick to avoid an unpredictable branch.
- Have several versions of loop for more restricted (but faster) cases; e.g. where all patterns are composed of free symbols with arity ≤ 3.
- Slot allocation: number slots independently on different branches, renumber slots using find-union so that variable can be bound from the same slot in each branch, renumber again using graph coloring to minimize number of slots.
- Slot storage allocated at compile time (matching is thus non-reentrant; causes various problems but much faster)

Free Theory Righthand Side Construction

- Naive recursive algorithm copies righthand replacing variables with their bindings.
- Better algorithm compiles righthand side into an automaton.
- Combine common subexpressions.
- Implementation trick: make substitution vector big enough to hold construction stack; then access to a variable binding is same as access to previously built node — save a test and branch at run time.
- Each instruction creates a new term graph node with a given symbol and arguments.

Congruence Class Rewriting

- Some equations cannot be oriented to give a terminating system (commutativity).
- Other equations can be oriented but it is convenient to use them both ways during a computation (associativity, identity, idempotence).
- Solution is to group these as a set E of axioms and rewrite on the congruence classes modulo E.
- Let $E = \{f(X, Y) = f(Y, X)\}$.

$$[f(a, g(f(b, c)))]_E = \{f(a, g(f(b, c))), f(a, g(f(c, b))),$$

$$f(g(f(b, c)), a), f(g(f(c, b)), a)\}$$

Motivating Example

```
fmod EXPR is
  sort Int .
  op 0 : -> Int .
  op -_ : Int -> Int .
  op _+_ : Int Int -> Int [assoc comm] .
  op _^_ : Int Int -> Int [assoc comm] .

vars X Y : Int .
  eq 0 + X = X .
  eq X + - X = 0 .
  eq X ^ 0 = X .
  eq X ^ X = 0 .
endm
```

Would like to be able to simplify:

```
(a + b + c + - a) ^ a ^ (b + c)
```

E-matching

- Congruence classes are too large to generate in practice.
- Congruence class rewriting can often be effective simulated by choosing a normal form for terms modulo E and using matching modulo E (possibly with extension) instead.
- Idea is to permute the pattern instead of the subject. Find one (or all) σ such that

$$[p\sigma]_E = [s]_E$$

Disjoint Theories

- E-matching for arbitrary E is of course undecidable.
- Practical approach is to allow $E = E_1 \cup \cdots \cup E_n$ where the theories $E_1 \ldots E_n$ are disjoint and each E_i is restricted to a set of axioms for which a matching algorithm is known.
- For almost all interesting theories E_i, matching is at least NP-hard if non-linear patterns are allowed.
- AC matching is NP-complete even for a single equation consisting of AC symbols on top of variables (pattern only) and constants.

Supported Theories 1

Maude allows the following axioms for a single binary function symbol f.

$f(X, Y) = f(Y, X)$	C	Commutativity
$f(f(X, Y), Z) = f(X, f(Y, Z))$	A	Associativity
$f(1, X) = X$	U_l	Left Unit (Identity)
$f(X, 1) = X$	U_r	Right Unit (Identity)
$f(X, X) = X$	I	Idempotence

Maude currently supports the following 17 combinations $(U = U_l + U_r)$:

$$C, CU, CI, CUI, A, AU_l, AU_r, AU, U_l, U_r, U,$$

$$I, U_lI, U_rI, UI, AC, ACU$$

Supported Theories 2

- Associativity-Commutativity (AC) and Associativity-Commutativity-Identity (ACU) correspond to multisets and are the most important combinations for practical applications and are the most heavily optimized in the engine. (Binary search, bipartite graph matching, inhomogeneous Diophantine equation solving.)
- Associativity (A) and Associativity-Identity (AU) correspond to lists or strings and are also fairly important. (Boyer-Moore, directed acyclic graph traversal, word partitioning.)
- Non-associative theories are much less useful; efficient (on average) algorithms are implemented but little special case optimization.

Unsupported Theories

- Theories that involve both Associativity and Idempotence are not yet implemented.
- ACI/ACUI correspond to sets.
- AI/AUI correspond to square free strings and have special problems.

Theory Normalization

- Associativity can be handled by flattening

$$f(a, f(f(b, c), d)) \equiv f(a, b, c, d).$$

- Commutativity can be handled by sorting. We need a total ordering on subterms in normal form. For example lexicographic ordering in the A case and multiset ordering in the AC case.
- Identity can be handle by collapsing out identity elements.
- Idempotence can sometimes be handled by collapsing out duplicates. (Doesn't work for AI for example.)

Extension

Sometimes a pattern p may match some subterm t' of $t \in [s]_E$ but not a subterm of s. For example if

$$E = \{f(X, f(Y, Z) = f(f(X, Y), Z)\}$$

$$p = f(g(X), h(Y))$$

$$s = f(g(a), f(h(b), i(c)))$$

Here in order to accurately simulate congruence class rewriting we need matching with extension - part of s is not matched. Can be achieved with extension variables or notion of extension can be built into matching algorithm.

3 Phase Matching 1

- Typically there may be many solutions to an E-matching problem.
- For conditional equations we want to generate them systematically.
- Want to use recursion to handle nested theories rather than exhaustive search.
- Want solutions returned sequentially from a generator object.

3 Phase Matching 2

The Maude interpreter rewrite engine handles E-matching in 3 phases.

Compile Pattern is compiled to a hierarchy of matching automata; structure of hierarchy follows structure of theories in pattern.

Match Matching automata is applied to subject to generate a hierarchy of subproblem objects which compactly represent all solutions.

Solve Solutions are extracted iteratively by iterative search algorithms for each theory.

Last phase is incremental; next solution is obtained by modifying current solution.

Constraint Propagation Analysis 1

- Order in which matching subproblems are solved is important if there are variables shared between them.
- Consider pattern

$$f(g(V,W), f(g(W,X), g(Y,Z), V), g(X,Y))$$

  with f free and g commutative.
- Naive approach is to search for solutions of the four commutative subproblems that are consistent on variable bindings.
- Better solution is to propagate bindings on variables.
- V can be bound uniquely once subject is known as its 2nd occurrence only has free symbols above it.

Constraint Propagation Analysis 2

- Once V is known we can find unique solutions for each subpattern in turn: $g(V, W)$ (binding W), $g(W, X)$ (binding X), $g(X, Y)$ (binding Y) and $g(Y, Z)$ (binding Z).
- With deep nesting of theories things can become more complicated.
- In general, compile phase figures out an optimal (in a particular sense) order in which to solve matching subproblems.

Greedy Matching 1

- Often patterns are very simple and we only need a single solution.
- Greedy algorithms work on a very restricted subset of patterns (restrictions depend on theory - also no sharing of variables between theories unless they are guaranteed already bound).
- Idea is to find a single solution as quickly as possible or show that no solution exists.
- Can occasionally return “undecided” — then we need to run the full algorithm.
- Greedy algorithms fully integrated with full algorithms; possible to match part of a pattern greedily and part use a full algorithm.

Greedy Matching 2

- Failing quickly is very important — most E-matching attempts fail.
- Consider the problem

$$f(g(X), h(Y), Z) \leq_E^? f(\alpha_1, \ldots, \alpha_n)$$

  with f AC, (AC terms are flattened). Can declare failure if nothing under f in the subject matches $g(X)$. Using binary search this can be very fast.
- Many-to-One AC matching algorithms have been proposed but in practice greedy matching seems to win.

E-rewriting on Large Terms 1

- AC/ACU terms are often used to represent large sets.
- A/AU terms are often used to represent long lists.
- In fact Maude encourages this since its predefined list/set/array/map data structures are implemented in Maude using this technique.
- Patterns used to match into these kind of subterms are often simple - typically pulling out one or more arguments.
- Uses of variables bound to this kind of subterms in righthand sides are often simple - typically adjoining one ore more arguments.
- Even greedy matching is too slow, since both the matching and replacement/normalization algorithms would have to touch hundreds or thousands of arguments.

E-rewriting on Large Terms 2

```
fmod LIST is
  sorts Elt List .
  subsort Elt < List .
  ops a b c d e : -> Elt .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil] .
  ops rev1 rev2 : List -> List .
vars E E' : Elt .
var L : List .
  eq rev1(nil) = nil .
  eq rev1(E L) = rev1(L) E .
  eq rev2(nil) = nil .
  eq rev2(E L E') = E' rev2(L) E .
endfm
```

E-rewriting on Large Terms 3

- Store flattened argument lists in persistent data structures and use special versions of the greedy matching algorithms and normalization algorithms.
- Supported algorithms strip off (by matching) one or two arguments and put the remaining arguments in a term that is bound to an unbound variable.
- Patterns stripping off an argument is either variable (bound or unbound) or an alien subpattern.
- Stripping of an argument during matching corresponds to deletion.
- Adjoining an argument during rhs construction/normalization corresponds to insertion.

E-rewriting on large terms 4

- AC/ACU uses persistent red-black trees to achieve $O(\log n)$ matching/normalization.
- A/AU uses persistent deques - to achieve amortized $O(1)$ matching/normalization (most of the time).
- Conversion of array based argument lists to persistent form is a heuristic in the theory normalization routines.
- Conversion of persistent form back to arrays is by need (only a few specialized algorithms handle the persistent form).

In-place Replacement 1

- In practice a term is represented as directed acyclic graph (dag) rather than a tree.
- When a shared dag node t rewrites we would like to see its replacement appear on all paths, not just on the path we came down.
- Replacement node might be bigger — more arguments for example.
- One alternative is an indirection table and replace pointers instead — expensive.
- In Maude rewrite engine all dag nodes are the same size. Part of the node is theory independent: virtual function pointer, symbol/copy pointer, sort info, flags.

In-place Replacement 2

Part of the node is theory dependent and can be used for storing:

- argument pointers (for small arity); or
- (base, used length, actual length) of an external argument list (may be variadic); or
- special non-term data (say a machine integer); or
- a pointer to an external object (say a bignum).

C++ allows a new object to be created, overwriting an old object.

Garbage Collection

- Each rewrite typically produces several dag nodes and these usually become garbage very quickly.
- Keep track of subset R of root pointers.
- Restrict garbage collects to safe-points (at least 1 per rewrite) when all in-use dag nodes are reachable from R.
- Mark reachable dag nodes and copy-compact argument lists.
- Incremental sweep — collect garbage dag nodes during allocation — churn through memory once per usage cycle.

Miscellaneous Issues

- Sort computation (ordered decision diagrams) and testing (byte vectors with short cuts)
- Stable symbol optimizations (binary searching, insertion in free theory net).
- Inter-theory collapse (collapse analysis, variable abstraction).
- Sharing and strategies (eager-laziness analysis, graph copying).
- Smaller sets of solutions for conditional equations.
- Left to right subexpression sharing.
- Special cheap-to-compute orderings on terms in theory normal form.
- Fancy subterm sorting algorithms using extra info.
- Maintaining (partial) normal form through matching.

Coherence

- Sometimes the term rewriting system T is not terminating and confluent.
- Split T in to equations E and rules R.
- E should be terminating and confluent.
- R need only be *coherent* wrt E.
- If $t \rightarrow_R t'$, $t' \rightarrow^!_E w$ and $t \rightarrow^!_E u$ then we require that there exists a step $u \rightarrow_R u'$ such that $u' \rightarrow^!_E w$.
- Idea is we can fully reduce using E after every R step without losing behavior/states.
- Maude has various execution mechanisms for R: search, LTL model checking, various notions of fairness, metalevel execution.

Maude Virtual Machine

- Building terms is expensive to compared to keeping pending evaluations on a stack.
- Implement a virtual machine for building righthand sides.
- Pending evaluations are stored as a stack of pointers into rhs instructions sequences.
- Terms are only built when no rule matches (i.e. fully reduced).
- Lots of implementation tricks possible; e.g. dispatch instructions using gcc's computed goto extension; specialized instructions/linear code for each small arity.
- Initial experiments give a $2\times$ speed up in the best case.
- Get back transparency (when needed) by having an alternative execution routine for pending instructions that constructs rather than evaluates to obtain (at some cost) the term that was virtually there but otherwise not explicitly constructed.

Recent Developments

- Narrowing - like rewriting but use unification rather than matching.
- Folded variant narrowing (controlled version of narrowing that carries extra information around - used for implementing new unification algorithms).
- Rewriting modulo SMT - a form of conditional rewriting when the side conditions are accumulated rather than evaluated (in general not executable). Conjunctions checked by an SMT solver (currently CVC4) solved to kill infeasible paths.