Electronic Notes in Theoretical Computer Science 4 (1996)

Fast matching in combinations of regular equational theories

S. Eker

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA

Abstract

We consider the problem of efficient term matching, modulo combinations of regular equational theories. Our general approach to the problem consists of three phases: compilation, matching and subproblem solving. We describe a technique for dealing with non-linear variables in a pattern and show how this technique is specialized to several specific equational theories. For matching in an order-sorted setting we discuss an important optimization for theories involving the associativity equation. Finally we sketch a new method of combining matching algorithms for regular collapse theories and give examples that involve the identity and idempotence equations.

1 Introduction

Rewriting logic provides a general way of specifying computational and logical systems [10,11]. An important part of this generality is making equational theories into an explicit parameter of the formalism. Since any computable data type can be equationally axiomatized [3], this allows specification of the data part of a system in a fully general way. We view a rewrite as acting on data structures so axiomatized.

Implementing rewriting logic for the purposes of executable specification presents interesting challenges, because of the generality of the equational theories that should ideally be supported. The present work has been motivated by the implementation of one such system — the Maude interpreter at SRI [4]. However, we believe that the techniques so developed are quite general and will be useful for other rewriting applications.

[1] Supported by a NATO Fellowship administered through the Royal Society, by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114 and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

While some part of an equational specification can be organized into a confluent and terminating set of rewrite rules, it is often convenient to have some equational theories handled implicitly by working with congruence classes of terms. For practical implementation purposes, we replace these congruence classes by chosen representatives and the matching step of term rewriting is performed by special matching algorithms, particular to the equational theories in use. For Maude it is important that these special matching algorithms be 'plug-compatible' so that the rewrite engine can be extended in a modular way as new matching algorithms are developed.

The general problem of combining matching algorithms for regular equational theories was solved by Nipkow [12]. While elegant from a theoretical standpoint, Nipkow's method, which is essentially based on variable abstraction, is far too inefficient for practical language implementation. In this paper we present selected algorithms and techniques developed for the Maude interpreter's rewrite engine. Experiments with the current version of the rewrite engine incorporating some of these techniques have shown speed-ups over the OBJ3 interpreter [8] of up to three orders of magnitude when rewriting with associative-commutative function symbols, even when all the patterns linear. When non-linear patterns are used speed-ups are more dramatic. On one associative-commutative rewriting problem the OBJ3 interpreter has failed to terminate after over 110 cpu-hours whereas the rewrite engine has succeeded in under 0.4 cpu-seconds running on the same hardware.

Since the matching algorithms we describe are imperative in nature and involve manipulating complex pointer based data structures, trying to force the algorithms into a functional form or presenting imperative pseudo-code would tend to obscure rather than illuminate the techniques involved. Instead we will focus on (abstract versions) of the data structures used and explain informally how they are constructed and manipulated with the aid of diagrams and carefully chosen examples.

1.1 Preliminaries

We work with the set of terms $T_\Sigma(\mathbf{X})$ over a signature Σ and a set $\mathbf{X}$ of variables. We denote the set of ground terms by T_Σ. We measure the size of a term t by counting the number $|t|$ of symbols in t. We denote the set of variables occurring in a term t by $Var(t)$.

An equation is said to be *regular* if every variable that appears on one side also appears on the other. An equation is *collapse-free* if neither side consists soley of a bare variable; otherwise it is a *collapse equation*. The most important regular collapse-free equations are those for associativity and commutativity for a given binary function symbol. The most important regular collapse equations are those for identity and idempotence for a given binary function symbol. A theory is *regular* if all its equations are regular. A theory is *collapse-free* if all its equations are collapse-free; otherwise it is a *collapse theory*.

For the purposes of combining matching algorithms we assume that the set of theories are disjoint; i.e., that no function symbol occurs in more than

one theory. Thus, each function symbol can be said to belong to a particular theory. When we have a term $f(\alpha_1, \ldots, \alpha_n)$ where the top symbol of a subterm α_i belongs to a theory other than that of f we call such α_i an *alien* subterm.

Let E be the union of all the theories containing symbols from Σ. Then we write $t =_E t'$ if $E \vdash t = t'$ by the deduction rules of equational logic. We denote the congruence class $\{t' \mid t' =_E t\}$ of t by $[t]_E$. We will write a matching problem in the combination E of theories as $p \leq^?_E s$, for $p \in T_\Sigma(\mathbf{X})$, $s \in T_\Sigma$ where p is called the *pattern* and s is called the *subject*. The goal of a matching problem is to find all distinct (gound) substitutions σ such that $p\sigma =_E s$. Two substitutions σ and σ' are considered distinct if for at least one variable X occurring in p, $\sigma(X) \neq_E \sigma'(X)$.

Our matching algorithms will build matching substitutions one piece at a time, and it will be useful to have the notion of a partial substitution. A partial substitution is a either a set $p \subset \mathbf{X} \times T_\Sigma$ satisfying the property

$$(X, t_1), (X, t_2) \in p \Rightarrow t_1 = t_2 \tag{1}$$

or the special constant *fail*. Let $\mathbf{P}$ be the set of all such partial substitutions. We define union on partial substitutions, $\sqcup : \mathbf{P} \times \mathbf{P} \to \mathbf{P}$ by

$$p \sqcup q = \begin{cases} p \cup q \text{ if } p \neq \mathit{fail} \text{ and } q \neq \mathit{fail} \text{ and } p \cup q \text{ satisfies (1)} \\ \mathit{fail} \quad \text{otherwise} \end{cases}$$

2 Regular collapse-free theories

If a subpattern p' has a top symbol belonging to some regular collapse-free theory E_1, then for any substitution σ, any $t \in [p'\sigma]_E$ will also have a top symbol belonging to E_1. This property means that if p' is an alien subpattern, it can only match some piece of the subject whose top symbol also belongs to E_1. Thus, we can recursively decompose a matching problem in a combination of regular collapse-free theories into smaller matching problems in individual theories for which we have matching algorithms.

We work with terms that have been normalized with respect to the union E of theories. Normalization can be seen as a function

$$\mathcal{NF} : T_\Sigma(\mathbf{X}) \to T_\Sigma(\mathbf{X})$$

that for each term t chooses a unique representative from its congruence class $[t]_E$. We assume that a total ordering $>$ on the set of symbols in Σ is given. For each equational theory, $\mathcal{NF}$ together with a total ordering on normal forms headed by the same function symbol (from that theory) must be defined. For terms that are just variable symbols or free constant symbols, $\mathcal{NF}$ is the identity function and the total ordering is the given ordering $>$ on symbols. For terms in normal form with differing top symbols, the total ordering on normal forms is given by the ordering on their top symbols.

Our basic approach to matching consists of three phases: compilation, matching and subproblem solving. We make the assumption that the same pattern p will be matched against many times with different subjects, and therefore it is worthwhile compiling it.

2.1 Compilation phase

In the compilation phase the normalized pattern is analyzed and a matching automaton is generated. We view compilation as a mapping

$$\mathit{compile} : T_\Sigma(\mathbf{X}) \times \mathcal{P}(\mathbf{X}) \to \mathbf{A}$$

where $\mathbf{A}$ is the set of *matching automata*. Matching automata are built up in a hierarchical way with larger automata containing subautomata. The actual form of a matching automaton depends on the equational theory in which the matching will be done; thus for an equational theory E_1 we will have E_1-automata. The hierarchical structure of a matching automaton closely reflects the structure of the pattern it was compiled from: where the p has a top symbol in theory E_1 and a pair of alien subterms in theories E_2 and E_3 for example, the compilation of p will be an E_1-automaton containing an E_2-automaton and an E_3-automaton.

The extra argument that *compile* takes is a set of variables whose bindings are guaranteed to be already known at match-time. For the top level compilation of a pattern this will usually be the empty set, however for the compilation of of subterms this extra argument allows some important optimizations as we shall see later. In all the theories we consider in this paper, the size matching of the automata for p will be linear in $|p|$.

2.2 Matching phase

In the matching phase the matching automaton produced from the compilation phase is applied to the normalized subject s. We view matching as a mapping

$$\mathit{match} : \mathbf{A} \times T_\Sigma \to \mathbf{P} \times \mathbf{S}$$

where $\mathbf{S}$ is the set of *subproblem objects*. Here the result of a match is a partial substitution which contains variables that can easily be determined to have the same value in all matching substitutions; together with a subproblem object which is a compact representation of the possible values for the variables not mentioned in the partial substitution. For a simple pattern the partial substitution might contain bindings for all the variables in the pattern in which case the empty subproblem object denoted by $\oslash$ is returned. Of course the matching phase could fail altogether in which case the pair $(\mathit{fail}, \oslash)$ is returned.

Subproblem objects are built up in a hierarchical way with larger subproblem objects containing subproblem sub-objects. As with matching automata the actual form of a subproblem object depends on the equational theory in which the matching was done; thus an E_1-automaton that could not uniquely bind all the variables occuring in the subpattern it was compiled from will construct an E_1-subproblem object to encode the alternatives. If the E_1-automaton contained an E_2-automaton then the E_1-subproblem object may contain one or more E_2-subproblem objects. In all the theories we consider in this paper, the matching phase requires at most $O(|p|.|s|)$ time and subproblem object has size at most $O(|p|.|s|)$.

2.3 Subproblem solving phase

For many simple patterns this phase will be unnecessary as the matching phase will have uniquely bound all the variables. For more complex patterns we are left with a partial substitution and a subproblem object which may contain nested subproblem sub-objects. In the subproblem solving phase the subproblem object is searched for consistent sets of solutions to the unbound variables; each such set corresponds to a different solution to the original matching problem. We view the searching process as a map

$$\mathit{solve} : \mathbf{P} \times \mathbf{S} \to \mathbf{P} \times \mathbf{S}$$

which takes a partial solution and a subproblem object and returns either a partial solution (which binds all the variables in the original pattern) and a new subproblem object which encodes the remaining possibilities, or the pair $(\mathit{fail}, \oslash)$ to indicate that there are no further matching substitutions.

Note that it is important that *solve* takes a partial substitution as its first argument. While at the top most level *solve* will always be called with the same partial substitution that was generated by *match*, at lower levels in the hierarchy of subproblem objects, *solve* will be called with different partial substitutions as the search progresses.

For implementation purposes subproblem objects actually contain state information to record which possibilities have already been tried and the returned subproblem object is really the original subproblem object with its state updated. Thus, solutions can be extracted from the subproblem object as needed.

In most of the theories we consider in this paper, matching with non-linear patterns is known to be NP-hard [2] and therefore the time required to find the next solution (or to discover that there is no next solution) may be exponential in the number of variables not bound during the matching phase.

2.4 Constraint propagation analysis

The order in which subterms are considered is very important for non-linear patterns. Consider for example the pattern

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

where f is a 3-ary free function symbol and g is a commutative function symbol. At first sight it might appear that, given a subject s, finding a matching substitution would require searching for solutions to four commutative matching subproblems that were consistent on the bindings to the non-linear variables V, W, X, Y. In fact, by propagating constraints on variables between the matching subproblems and by considering the subproblems in an optimal order, no searching is necessary. The variable V can be bound uniquely once the subject s is known, as its second occurrence has only free function symbols above it. Once the value of V is known the commutative matching subproblems can be solved directly in the order $g(V, W)$ (binding W uniquely), $g(W, X)$ (binding X uniquely), $g(X, Y)$ (binding Y uniquely) and $g(Y, Z)$ (binding Z uniquely).

This is an example of a general technique applicable to non-linear matching problems which can be used to propagate constraints between variable instances lying in different theories. During the compilation phase we find an 'optimal' order in which to match alien subterms by a process we call *constraint propagation analysis*. Constraint propagation analysis can take time that is exponential in $|p|$.

Constraint propagation analysis makes use of two functions that are closely related to *compile* and which are defined for each equational theory. These are

$$\mathcal{CPA} : T_\Sigma(\mathbf{X}) \times \mathcal{P}(\mathbf{X}) \to \mathcal{P}(\mathbf{X})$$

and

$$\mathcal{EMF} : T_\Sigma(\mathbf{X}) \times T_\Sigma(\mathbf{X}) \to \mathbf{B}.$$

Intuitively $\mathcal{CPA}(p, V)$ ("Constraint Propagation Analysis") is the union of V and the set of variables that the matching automaton $\mathit{compile}(p, V)$ would bind uniquely at match-time regardless of the subject. For instance in the above example $\mathcal{CPA}(g(V, W), \{V\}) = \{V, W\}$. For all the matching algorithms we consider in this paper $\mathcal{CPA}$ has the following very useful monotonicity property:

$$\mathcal{CPA}(t, V_1) \cup \mathcal{CPA}(t, V_2) \subseteq \mathcal{CPA}(t, V_1 \cup V_2).$$

Intuitively $\mathcal{EMF}(p, t)$ ("Early Match Fail") returns true if for any set of variables V and any substitution σ, $\mathit{match}(\mathit{compile}(p, V), t\sigma) = (\mathit{fail}, \oslash)$; in other words any automaton for p will always return failure when attempting to match any instance of t. Note that this is stronger than saying that p and t are not unifiable (assuming that $\mathit{Var}(p) \cap \mathit{Var}(t) = \emptyset$). In practice this function can be defined in a conservative way; it is always okay for $\mathcal{EMF}$ to return false and as worst we may miss a potential optimization in the construction of an automaton. An easy way to defined $\mathcal{EMF}$ would be $\mathcal{EMF}(p, t)$ is true if and only if p and t have top symbols in different collapse-free theories. Because of this we will not consider the definition of $\mathcal{EMF}$ any further. We will say that two patterns p_1 and p_2 are *match-independent* if and only if $\mathcal{EMF}(p_1, p_2)$ and $\mathcal{EMF}(p_2, p_1)$ are both true (the definition of $\mathcal{EMF}$ need not be commutative).

3 Particular theories

We now informally describe the normal forms, orderings, automata, constraint propagation functions and subproblem objects used for some particular theories.

3.1 Free theory

For a term headed by a free function symbol $f_\emptyset$ we define the normal form by

$$\mathcal{NF}(f_\emptyset(\alpha_1, \ldots, \alpha_n)) = f_\emptyset(\mathcal{NF}(\alpha_1), \ldots, \mathcal{NF}(\alpha_n)).$$

If $\alpha = f_\emptyset(\alpha_1, \ldots, \alpha_n)$ and $\beta = f_\emptyset(\beta_1, \ldots, \beta_n)$ are in normal form then $\alpha > \beta$ iff there exists $i \in \{1, \ldots, n\}$ such that $\alpha_i > \beta_i$ and for $j \in \{1, \ldots, i-1\}$,

$\alpha_j = \beta_j$. Because the free theory has no equations, all free functions symbols can be considered to belong to the same theory and a single automaton can be used to match a whole free skeleton; i.e. that part of a term obtained by ignoring all subterms headed by non-free symbols. Consider a pattern

$$t = c[X_1, \ldots, X_n, \alpha_1, \ldots, \alpha_m]$$

where c is a free skeleton, $X_1, \ldots, X_n$ are variables and $\alpha_1, \ldots, \alpha_m$ are alien subterms. We now sketch the form of $\mathit{compile}(t, V)$ and $\mathcal{CPA}(t, V)$. Let $\phi : \{1, \ldots, m\} \to \{1, \ldots, m\}$ be a permutation. We define

$$V_0^\phi = V \cup \{X_1, \ldots, X_n\},$$
$$V_i^\phi = \mathcal{CPA}(\alpha_{\phi(i)}, V_{i-1}^\phi) \text{ for } i \in \{1, \ldots, m\}.$$

Intuitively V_m^ϕ is the set of variables that could be uniquely bound if we built our automaton so as to match the aliens subterms in the order given by the permutation ϕ. We can compare the merits of two permutations ϕ and ψ by comparing $|V_m^\phi|$ and $|V_m^\psi|$. In principle we can find an 'optimal' permutation by searching through all permutations to find one which uniquely binds at least as many variables as any other. In practice the monotonicity property of $\mathcal{CPA}$ allows us to dramatically prune the search space in most cases.

Having selected an optimal permutation ϕ we can put $\mathcal{CPA}(t, V) = V_m^\phi$ and generate a sequence $A_1, \ldots, A_m$ of subautomata for $\alpha_{\phi(1)}, \ldots, \alpha_{\phi(m)}$ by

$$A_i = \mathit{compile}(\alpha_{\phi(i)}, V_{i-1}^\phi) \text{ for } i \in \{1, \ldots, m\}.$$

Then the automaton $\mathit{compile}(t, v)$ will consist of:

(i) a sequence of positions (relative to the top of the free skeleton) and free symbols to check;

(ii) a sequence of positions and variables $(X_1, \ldots, X_n)$ to bind (or check for a clash if already bound); and

(iii) a sequence of positions and alien automata $(A_1, \ldots, A_n)$ to apply to the subject.

Since each of the alien automata could return a subproblem object, a free theory subproblem object consists of a sequence of alien subproblem objects to be solved in order with backtracking on failure.

3.2 Commutative theories

For a term headed by a commutative function symbol f_C we define the normal form by

$$\mathcal{NF}(f_C(\alpha_1, \alpha_2)) = \begin{cases} f_C(\mathcal{NF}(\alpha_1), \mathcal{NF}(\alpha_2)) \text{ if } \mathcal{NF}(\alpha_1) > \mathcal{NF}(\alpha_2) \\ f_C(\mathcal{NF}(\alpha_2), \mathcal{NF}(\alpha_1)) \text{ otherwise} \end{cases}$$

If $\alpha = f_C(\alpha_1, \alpha_2)$ and $\beta = f_C(\beta_1, \beta_2)$ are in normal form then $\alpha > \beta$ iff either $\alpha_1 > \beta_1$ or $\alpha_1 = \beta_1$ and $\alpha_2 > \beta_2$.

Consider a pattern $t = f_C(\alpha_1, \alpha_2)$ We now sketch the form of $\mathit{compile}(t, V)$ and $\mathcal{CPA}(t, V)$. There are several cases to consider:

Suppose that $Var(\alpha_1) \subseteq V$. Then at match-time, since all the variables in V will already be bound we know exactly what α_1 will match against. Thus the uncertainly introduced by the commutativity equation is removed and we can put $\mathcal{CPA}(t, V) = \mathcal{CPA}(\alpha_2, V)$. A symmetric case occurs if $Var(\alpha_2) \subseteq V$.

Suppose that α_1 and α_2 are match-independent aliens. Then α_1 and α_2 will only be able to match to corresponding aliens in the subject in at most one way and we can discover which way at match-time since when we try the wrong way we are guaranteed failure without having to wait until the subproblem solving phase. Thus the uncertainty introduced by the commutativity equation is removed and we have a choice of two values for $\mathcal{CPA}(t, V)$:

$$\mathcal{CPA}(\alpha_2, \mathcal{CPA}(\alpha_1, V)) \text{ or } \mathcal{CPA}(\alpha_1, \mathcal{CPA}(\alpha_2, V))$$

depending on which of the aliens we match first; we chose the order that gives us the largest set of uniquely bound variables.

If the above cases do not apply then we cannot guarantee to uniquely bind any additional variables during the match phase and we have $\mathcal{CPA}(t, V) = V$.

A commutative theory automaton consists of:

(i) the top symbol to check (f_C);

(ii) a sequence of at most two variables; and

(iii) a sequence of at most two alien automata.

A commutative theory subproblem object consists of at most two sequences of alien subproblem objects (one for each possible way around to match the alien patterns against alien subjects), each of which is of length at most two and is accompanied by a local partial substitution.

3.3 Associative theories

In order to construct a convenient normal form for terms in associative theories we allow associative function symbols to be variadic. For a term headed by an associative function symbol f_A we define the normal form by

$$\mathcal{NF}(f_A(\alpha_1, \alpha_2)) = f_A(\alpha_1', \alpha_2')$$

where

$$\alpha_i' = \begin{cases} \gamma_1, \ldots, \gamma_n \text{ if } \mathcal{NF}(\alpha_i) = f_A(\gamma_1, \ldots, \gamma_n) \text{ for some } \gamma_1, \ldots, \gamma_n \\ \mathcal{NF}(\alpha_i) \quad \text{otherwise} \end{cases}$$

If $\alpha = f_A(\alpha_1, \ldots, \alpha_n)$ and $\beta = f_A(\beta_1, \ldots, \beta_m)$ are in normal form then $\alpha > \beta$ iff there exists $i \in \{1, \ldots, n\}$ such that for $j \in \{1, \ldots, \min(i-1, m)\}$, $\alpha_j = \beta_j$ and either $i > m$ or $\alpha_i > \beta_i$.

Consider a term $t = f_A(\alpha_1, \ldots, \alpha_n)$. We now sketch the form of $compile(t, V)$ and $\mathcal{CPA}(t, V)$. The key idea is to partition the sequence $\alpha_1, \ldots, \alpha_n$ of subterms into three subsequences, r_1, q, r_2 where r_1 and r_2 will be collectively called the *rigid part* and q will be called the *flex part*. The intuition is that at match time, given a subject $f_A(s_1, \ldots, s_m)$, for each α_i in the rigid part we will be able to determine uniquely what part of the subject it matches.

The partitioning of $\alpha_1, \ldots, \alpha_n$ into rigid and flex parts is complicated and we will not attempt to define it here. Instead we give a concrete example that illustrates some of the subtleties involved. Consider

$$t = f_A(g(X, Y), Y, h(A), Z, h(B), W, h(X))$$

where g is a commutative function symbol and h is a free function symbol. Then the rigid part will consist of $r_1 = g(X, Y), Y, h(A)$ and $r_2 = h(X)$ and the flex part will consist of $q = Z, h(B), W$. The reasoning behind this partitioning is as follows: Consider matching a subject $f_A(s_1, \ldots, s_m)$. We know that $g(X, Y)$ must match s_1 and $h(X)$ must match s_m. Furthermore if we match $h(X)$ we will uniquely bind X (since $\mathcal{CPA}(h(X), \emptyset) = \{X\}$) and then matching $g(X, Y)$ we will uniquely bind Y (since $\mathcal{CPA}(g(X, Y), \{X\}) = \{X, Y\}$). Once we have a binding for Y we know which s_i must match $h(A)$ and we can uniquely bind A. At this point we cannot make any further guarantees about what variables will be uniquely bound at match-time since Z and W could match any number of alien subjects and there may be a whole set of solutions. In this example we assumed an empty set of variables with guaranteed unique bindings at the outset so we have

$$\mathcal{CPA}(t, \emptyset) = \{A, X, Y\}.$$

If we knew that W would be uniquely bound at match time (i.e. if t was part of a larger pattern that would uniquely bind W) then the rigid part would consist of the entire sequence $g(X, Y), Y, h(A), Z, h(B), W, h(X)$ and the flex part would be empty. Consequently

$$\mathcal{CPA}(t, \{W\}) = \{A, B, W, X, Y, Z\}.$$

An associative automaton consists of:

(i) the top symbol to check (f_A);

(ii) length bounds for fast failure tests;

(iii) the compiled rigid part; and

(iv) the compiled flex part.

Both the rigid and flex part compile to a sequence of variables and alien automata. The the compiled rigid part contains additional information about what order its members should be matched in. As a further optimization sequences of consecutive alien terms in the flex part may be analyzed using the $\mathcal{EMF}$ function to calculate a shift table similar to one of those used in the Boyer-Moore string matching algorithm.

In general the flex part may match the remaining part of the subject in may ways and generate an associative subproblem object. We now illustrate the structure of an associative subproblem object with the following example. Consider the flex part $q = W, \alpha, X, \beta, Y, \gamma, Z$. Let $\alpha_1, \alpha_2, \alpha_3$ be subterms matching α, $\beta_1, \beta_2, \beta_3, \beta_4$ be subterms matching β, $\gamma_1, \gamma_2, \gamma_3$ be subterms matching γ and $\delta_1, \delta_2, \delta_3$ be other alien subterms. Suppose after matching to rigid part the following sequence of subject terms is left:

$$\delta_1, \alpha_1, \alpha_2, \beta_1, \beta_2, \beta_3, \gamma_1, \alpha_3, \gamma_2, \beta_4, \delta_2, \gamma_3, \delta_3$$

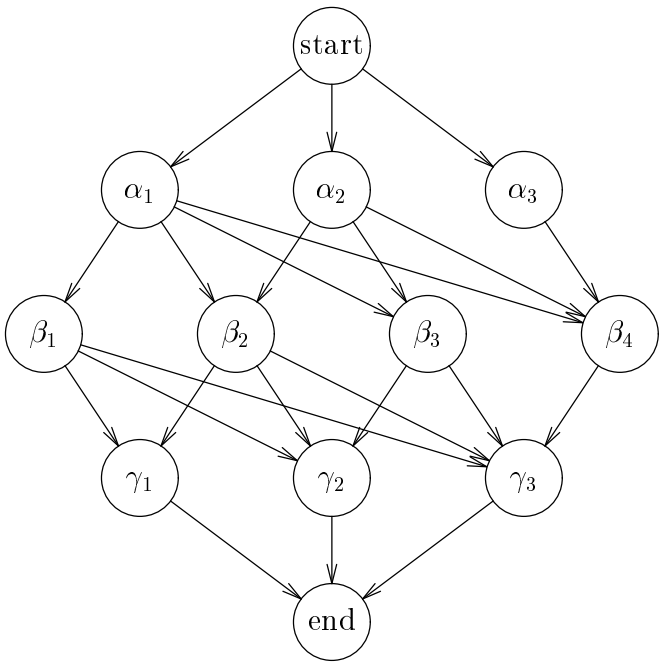

Fig. 1. Dag representing associative subproblem

Notice that if α is matched against α_1 then it is possible for β to be matched against any of $\beta_1, \beta_2, \beta_3$ where as if α is matched against α_2 then matching β against β_1 is not possible as X must be assigned at least one subject. In general the possible combinations of matches for alien patterns α, β, γ can be compactly represented as a directed acyclic graph (dag) as shown in Fig. 1.

Here the node α_1 for example will hold the local partial substitution and subproblem object generated by matching α against α_1. Each potential solution to the associative matching problem corresponds to a path from *start* to *end* in the graph. Notice that dags representing matching problems in this way have a special form; for a given patten α if the node representing some matching subject α_j has an arc going to some node n then the node representing any earlier matching subject α_i will have an arc going to n. This property allows the alternative more compact representation shown in Fig. 2.

Again each path through the dag from *start* to *end* represents a potential solution to the associative matching problem. However here every node has at most two arcs, a horizontal and a vertical one. Taking a vertical arc corresponds to committing to an alien subject (represented by the source node) for some alien pattern where as taking a horizontal arc corresponds to skipping that subject. This structure and *solve* algorithm for it is similar to that described in [5]. Note that in this example we have made the simplifying assumption that the flex part consists of alternating variables and alien subterms.

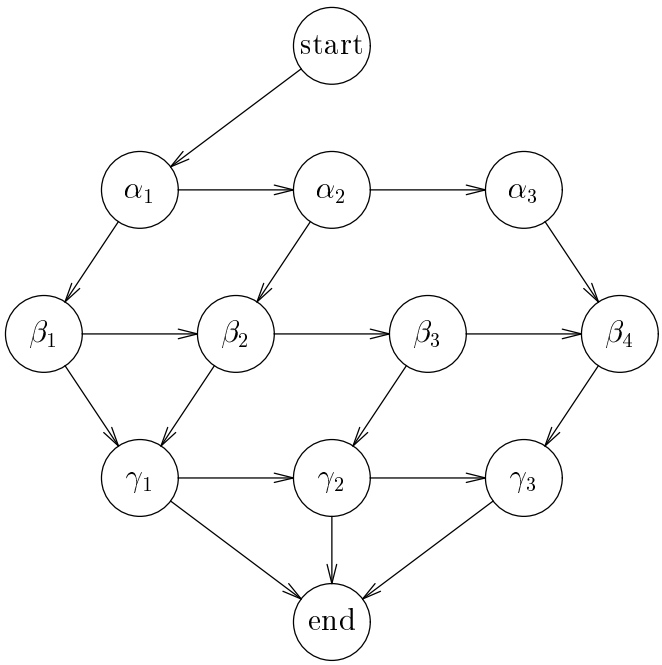

Fig. 2. Simplified dag representing associative subproblem

3.4 Associative-Commutative theories

We use the *ordered normal form* and term ordering introduced by Hullot [9]. Here, we allow associative-commutative function symbols to be variadic, and allow the arguments of an associative-commutative function symbol to take a positive integer *multiplicity* denoted by a superscript. The normal form is obtained by flattening nested occurences of the same associative-commutative function symbol (as we did for associative function symbols), normalizing and sorting the alien subterms, and replacing k identical subterms by a single instance with multiplicity k. For example consider

$$t = f_{AC}(f_{AC}(\alpha_4, \alpha_1), f_{AC}(\alpha_3, \alpha_2))$$

where each α_i is headed by a symbol other than f_{AC}, $\mathcal{NF}(\alpha_2) = \mathcal{NF}(\alpha_4)$ and $\mathcal{NF}(\alpha_1) > \mathcal{NF}(\alpha_2) > \mathcal{NF}(\alpha_3)$. Then $\mathcal{NF}(t)$ is

$$f_{AC}(\mathcal{NF}(\alpha_1), (\mathcal{NF}(\alpha_2))^2, \mathcal{NF}(\alpha_3))$$

The ordering on two terms in normal form which are headed by the same associative-commutative function symbol is given by the multiset ordering on their subterms.

Consider the pattern $t = f_{AC}(\alpha_1^{k_1}, \ldots, \alpha_n^{k_n})$ We now sketch the form of $compile(t, V)$ and $\mathcal{CPA}(t, V)$. Since the associative and commutative equations together allow arbitrary permutation of the subterms under an associative-commutative function symbol the possibilities for constraint propagation analysis are severely limited. There are however some special cases.

Suppose that for $i \in \{2, \ldots, n\}$, $Var(\alpha_i) \subseteq V$. Then at match-time we will know exactly what subject terms $\alpha_2^{k_2}, \ldots, \alpha_n^{k_n}$ will match; and once these have been eliminated, whatever is left must match $\alpha_1^{k_1}$. Thus $\mathcal{CPA}(t, V) = CPA(\alpha_1, V)$. There are of course $n - 1$ other symmetric cases.

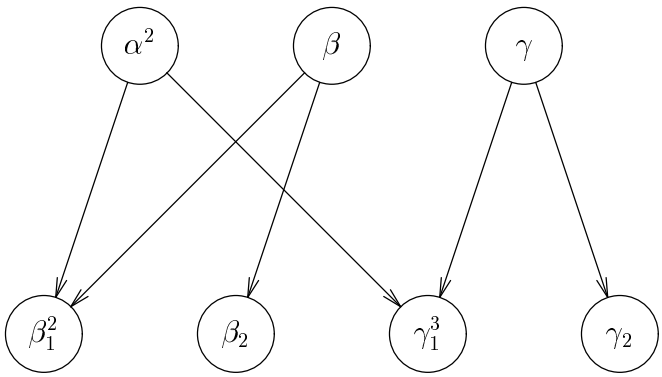

Fig. 3. Bipartite graph representing associative-commutative subproblem

Suppose that none of the α_i is a variable. Then each α_i will match exactly one alien subterm in the subject. Now if some α_i is match-independent from all the others then we will be able to find a unique match for it (if it matched more than one alien subterm in the subject, match independence would ensure that the match as a whole would fail). We call such an α_i *match-unique*. Now if we take the set of all match-unique subpatterns we can use the same search technique that we described for the free theory to find an optimal order to match them and do constraint propagation analysis.

An associative-commutative automaton consists of:

(i) the top symbol to check;

(ii) multiplicity bounds for fast failure tests;

(iii) a set of variables (with multiplicities);

(iv) a set of ground alien terms (with multiplicities); and

(v) a sequence of alien automata (with multiplicities).

At match-time ground alien terms can be rapidly cancelled with alien terms in the subject by using binary search. A similar technique can be used with the values of any bound variables.

We now illustrate the structure of an associative-commutative subproblem object. Suppose that after eliminating ground terms and bound variables we are left with the variables X^2, Y and the alien subterms α^2, β, γ in the pattern and the aliens subterms $\beta_1^2, \beta_2, \gamma_1^3, \gamma_2$ in the subject. Let β match β_1 and β_2; γ match γ_1 and γ_2; and α match all four subject subterms. This situation can be compactly represented by the bipartite graph shown in Fig. 3 and the set of variables (with multiplicities).

Here the arc from the node α^2 to the node β_1^2 will be labelled with the local partial substitution and the subproblem object generated by matching α against β_1 for example. Note that there is no arc from node α^2 to node β_2 because of multiplicity considerations.

An associative-commutative subproblem object consists of such a bipartite graph and a set of variables (with multiplicities). It is solved by a modified form of bipartite graph solving together with Diophantine system solving (to handle the remaining unbound variables). A more detailed description of a

similar associative-commutative-matching algorithm (but omitting the compilation phase and including separate Diophantine system solving phase) is given in [6].

4 An optimization for order sorted matching

We now give an important optimization technique for matching modulo associativity (or associativity-commutativity) in an order sorted setting. Consider the following pseudo-Maude code to define a function f on an associative list.

```
fmod LIST is
  sorts Element List .
  subsort Element < List .
  op cons : List List -> List [assoc] .
  op f : List -> List .

var E : Element .
var L : List .
  eq f(E) = ...
  eq f(cons(E, L)) = ...
endfm
```

Here the subpattern $cons(E, L)$ is used to extract a single element from the head of a list. Consider the subject $f(cons(a, b, c, d))$. If sorts are tested as substitutions are generated, we could be unlucky and generate the sequence of substitutions $\{E \to cons(a, b, c), L \to d\}, \{E \to cons(a, b), L \to cons(c, d)\}, \{E \to a, L \to cons(b, c, d)\}$. The first two substitutions fail because E can only take terms of sort `Element` and any term with *cons* has sort `List` which is larger. Naturally, generating the superfluous substitutions can get extremely expensive for large lists. A similar problem arises with associative-commutative set constructors. While it is possible to solve the problem for simple patterns using ad hoc tricks we present a general method.

Since we are interested in associative (and associative-commutative) function symbols, the domain and range sorts must lie in the same connected sort component and thus, for simplicity, we can assume that the sort structure consists of just a single connected sort component S.

If an order-sorted signature is *pre-regular* [7], for each n-ary function symbol f there exists a order-sorting function

$$\mathcal{S}_f : S^n \to S$$

which given a tuple of argument sorts $s_1, \ldots, s_n$ yields the least sort of any term $f(\alpha_1, \ldots, \alpha_n)$ where α_i has least sort s_i.

Let f be is an associative (associative-commutative). We insist that $\mathcal{S}_f$ is associative (associative-commutative) otherwise there may not be a well defined least sort for every congruence class. There exists a function

$$\mathcal{B}_f : S \to \{1, 2, 3, \ldots\} \cup \{\infty\}$$

for $s \in S$ **do** $\mathcal{B}_f[s] := \infty$ **od**;
$n = 1$; *limit* $:= 1$;
while $n \leq$ *limit* **do**
    $T = \emptyset$;
    for $(s_1, s_2) \in S^2$ **do**
        if $\mathcal{B}_f[s_1] + \mathcal{B}_f[s_2] > n$ **then** $T := T \cup \{s \mid s \leq \mathcal{S}_f[s_1, s_2]\}$ **fi**
    od;
    for $s \in S$ **do**
        if $s \notin T \wedge \mathcal{B}_f[s] = \infty$ **then** $\mathcal{B}_f[s] := n$; *limit* $:= 2 \times n$ **fi**
    od;
    $n := n + 1$
od

Fig. 4. Algorithm for computing $\mathcal{B}_f$ from $\mathcal{S}_f$.

which for each sort s gives an upper bound on the number of alien subterms that could be assigned to a variable of sort s occurring under f in a pattern. In the above example $\mathcal{B}_{cons}(\texttt{Element}) = 1$ and $\mathcal{B}_{cons}(\texttt{List}) = \infty$. In order to precisely define $\mathcal{B}_f$ we first extend $\mathcal{S}_f : S^2 \to S$ to a family of functions $\mathcal{S}_f : S^n \to S$ for $n \geq 2$ by the following recursive definition.

$$\mathcal{S}_f(s_1, \ldots, s_n) = \mathcal{S}_f(\mathcal{S}_f(s_1, \ldots, s_{n-1}), s_n) \text{ for } n > 2.$$

Then $\mathcal{B}_f(s)$ is the least n such that for all $\vec{s} \in S^n$, $\mathcal{S}_f(\vec{s}) \not\leq s$ if such an n exists or ∞ otherwise. To effectively compute $\mathcal{B}_f$ in polynomial time we represent both $\mathcal{S}_f$ and $\mathcal{B}_f$ as arrays indexed by sorts and use the algorithm given in Fig. 4.

The function $\mathcal{B}_f$ can be used during associative (or associative-commutative) matching to avoid generating useless assignments to variables under f which have sort s for some s with $\mathcal{B}_f(s) \neq \infty$. Also when compiling a pattern $f(t_1, \ldots, t_n)$ for an associative function symbol f and subterm t_i which is a variable of sort s for $\mathcal{B}_f(s) = 1$ may be treated as an alien for the purposes of determining the rigid part and constraint propagation since it will only match a single alien in the subject. In the example above the subpattern $cons(E, L)$ compiles to a particularly efficient associative matching automaton which never generates a subproblem object.

5 Regular collapse theories

We start with the observation that regular collapse equations have a very restricted form; one side is some bare variable X and the other side may not contain variables other than X. For simplicity we will only consider the two most important collapse equations; identity and idempotence for a given binary function symbol. We will assume than normal forms chosen for collapse theories are such that terms in normal for are fully collapsed; i.e. no application of a collapse equation in the collapse direction is possible. In many cases designing individual matching algorithms for regular collapse theories

are not much more difficult than designing algorithms for regular collapse-free theories. In particular, the algorithms for associative and associative-commutative matching can easily be extended to handle associative-identity and associative-commutative-identity matching respectively by allowing the possibility of variables not being assigned any alien subject subterm, but instead taking the identity element. (Although in some other collapse theories, notably associative-idempotent and associative-idempotent-identity theories even computing unique normal forms can be tricky, in this case because of the lack of commutativity.) The real problem arises with the combination of collapse theories. Consider for example the matching problem

$$f(X, u(Y, f(X, Y))) \leq^?_E f(a^2, 1_u)$$

where f is associative-commutative and u has the identity element 1_u. The straightforward decomposition into smaller matching problems which we used for collapse-free theories fails here, because of what we call an *inter-theory collapse*; when Y is assigned 1_u, the subpattern $u(Y, f(X, Y))$ collapses to $f(X, 1_u)$ whose top symbol belongs to the enclosing theory.

The OBJ3 interpreter gets around this problem by adding additional conditional equations to the rewrite system using a process called *identity completion*. Our approach is to do bottom-up preprocessing on the patterns before compiling them, making explicit the possible inter-theory collapses by the generation and propagation of *OR-nodes*.

5.1 OR-nodes

We consider terms as ordered trees, with nodes labeled by elements of $\Sigma \cup \mathbf{X}$ and arcs unlabelled. Preprocessing will introduce OR-nodes into the tree. An OR-node is a special node that has two or more labelled arcs to alternative subtrees — intuitively these represent different possible collapses of a given subpattern while the labels on the arcs represent the conditions necessary for the collapses to take place. Each OR-node arc label consists of a (possibly empty) conjunction of matching problems $p_1 \leq^?_E s_1 \wedge \cdots \wedge p_n \leq^?_E s_n$. The result of matching one of the alternative subtrees of an OR-node is only valid if the conjunction of matching problems on the corresponding arc can be solved (binding the variables that were eliminated by the collapse). OR-nodes are introduced into a the tree representing a pattern when the possibility of a collapse is recognized. An OR-node is propagated up the tree when one of its subtrees interacts with (i.e. has its top fucntion symbol in the same theory as that of) its parent node.

The preprocessed pattern forms an *OR-tree*. When an OR-tree is compiled, the OR-nodes compile to OR-automata which have a subautomaton for the subterm below each arc together with additional subautomata to deal with the conjunctions of matching problems labelling the arcs. At match-time OR-automaton may give rise to OR-subproblem objects. We now illustrate the technique with two examples.

First we consider the pattern $f(X, u(Y, f(X, Y)))$ from the matching problem example above. Working bottom-up $f(X, Y)$ cannot collapse so prepro-

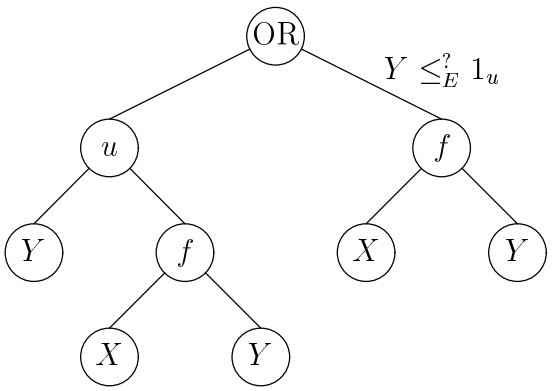

Fig. 5. OR-tree for $u(Y, f(X, Y))$

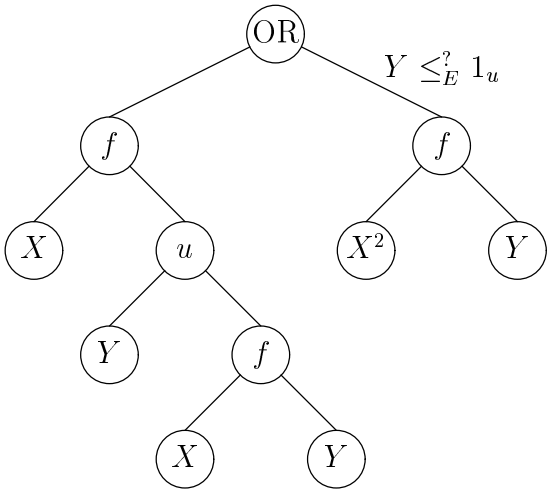

Fig. 6. OR-tree for $f(X, u(Y, f(X, Y)))$

cessing leaves it unchanged. The possibility of a collapse occurs with the subpattern $u(Y, f(X, Y))$. Preprocessing introduces an OR-node with two alternatives as shown in Fig. 5.

In the left alternative no collapse happens and the subpattern remains unchanged. The right alternative is where $Y = 1_u$ and the subpattern collapses to $f(X, Y)$. The condition $Y = 1_u$ appears as a matching problem on the right arc; although here the condition is particularly simple and does not need the full generality of matching. Moving up the pattern we see that there is the possibility of an interaction between the top f function symbol and the f in the right alternative of the OR-node, so the OR-node must be propagated upwards, yielding the OR-tree shown in Fig. 6.

In the left alternative in Fig. 6 where no collapse takes place and the original pattern appears below the arc. The right alternative is where the collapse takes place and the two f symbols combine by flattening; the condition on the right arc of the OR-node has not changed.

We now give a more complex example which illustrates the need for conjunctions of matching problems labelling the OR-node arcs. Consider the pattern $f(a, i(f(a, Z), f(b, X), Y), X)$ where where f is associative-commutative

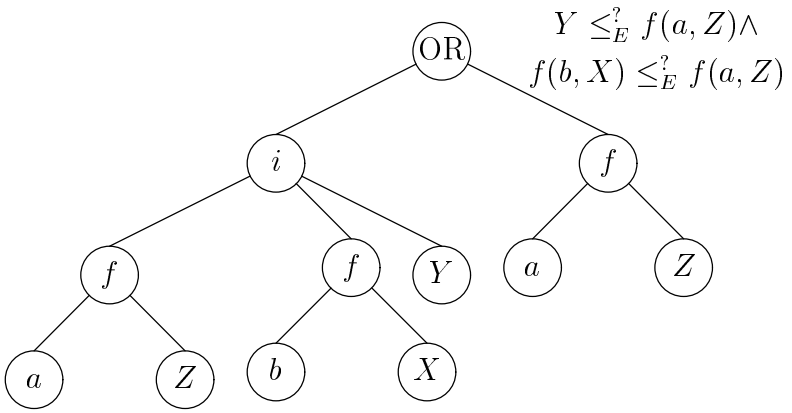

Fig. 7. OR-tree for $i(f(a,Z), f(b,X), Y)$

and i is associative-commutative-idempotent. Working bottom-up the possibility of a collapse occurs with the subpattern $u(Y, f(X,Y))$. Preprocessing introduces an OR-node with two alternatives as shown in Fig. 7.

The left alternative is where no collapse happens and the subpattern remains unchanged. The right alternative is where all three subpatterns under the symbol i match the same thing and the subterm headed by i collapses by the idempotence equation. Since in this case all three subpatterns under i match the same thing we can arbitrarily choose which one the subterm headed by i collapses to; we choose $f(a,Z)$. The condition for the collapse to take place is that $Y =_E f(a,Z)$ and $f(b,X) =_E f(a,Z)$. At first sight it may appear that we have introduced a unification problem with this condition. However at match-time when this condition is tested $f(a,Z)$ will already have been matched and Z will be bound to a ground term and hence we can treat $f(a,Z)$ as ground and evaluate the condition using only matching.

In general when a pattern $i(p_1,p_2)$ collapses p_1 due to idempotence during the preprocessing step there are a couple of subtle issues. Firstly since p_1 will ultimately be matched against (part of) the subject (even though it may then be part of a bigger pattern), all the variables in it will ultimately be bound. However they may not be bound uniquely until the subproblem solving phase and thus the condition $p_2 \leq^?_E p_1$ may have to be pushed into an OR-subproblem object. Secondly p_1 and p_2 themselves may already be OR-trees. In this case the collapse branches of p_1 may be discarded in the condition since they correspond to normalization of p_1 after substituting for its variables which can be done by our normal form function.

Moving up the pattern we see that there is the possibility of an interaction between the top f function symbol and the f in the right alternative of the OR-node, so the OR-node must be propagated upwards, yielding the OR-tree shown in Fig. 8.

Here the left alternative of the OR-node is where no collapse takes place and the original pattern appears below the arc. The right alternative is where the collapse takes place and the two f symbols combine by flattening; the condition on the right arc of the OR-node has not changed.

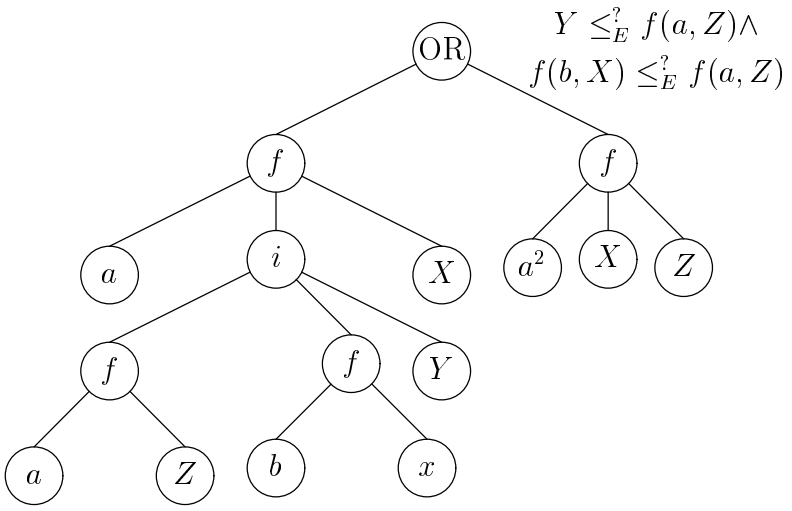

Fig. 8. OR-tree for $f(a, i(f(a, Z), f(b, X), Y), X)$

6 Concluding Remarks

We have presented a general approach to matching in combinations of regular equational theories. The key ideas are:

(i) To analyze the pattern before the subject is seen in order to determine the best order to match its parts and to produce an automaton which includes additional information to accelerate the matching process.

(ii) To have a polynomial time match phase which binds those variables for which it can find unique bindings and constructs a subproblem object to encode the possible bindings of the other variables.

(iii) To confine the potentially exponential searching needed to find consistent sets of solutions for smaller matching problems to a subproblem object whose design is optimized for fast non-recursive exploration of the search space.

(iv) To make use of the sort structure of associative (associative-commutative) function symbols to prune the search space.

(v) To preprocess patterns to make inter-theory collapses explicit.

We notice that the kind of matching problems that arise in algebraic (OBJ-style) programming often differ considerably from those that arise in theorem proving (e.g. using structural induction). The former tend to have simple linear or almost linear patterns for which constraint propagation analysis is useless but huge associative (associative-commutative) subjects (representing data). The latter tend to have complex non-linear patterns but relatively small subjects. Both kinds of matching problems can present difficulties to naive matching algorithms. The algorithms we have discusses are mostly aimed at the latter kind of matching problem. The current version of the rewrite engine lacks collapse theories but includes a second family of algorithms that we call *greedy matching algorithms* that are optimized for generating a single solution to matching problems that have a simple pattern. These provide a small (but useful) constant factor speed up in many cases.

Two problems that we have yet to considered are many-to-one matching and order of generation of matching substitutions. Many-to-one matching is important in term rewriting because one has a single subject and a choice of rewrite rules. The indexing scheme currently used in the rewrite engine is based on [13] and gives the equivalent of many-to-one for the topmost free function symbol skeleton (if any) of each pattern. An algorithm for many-to-one matching in the linear associative-commutative case is given in [1]. Going beyond this to combine many-to-one matching with constraint propagation analysis seems particularly difficult.

For many (confluent, terminating) term rewriting systems the choice of matching substitutions used has a marked effect on the number of rewrites needed to bring a term into fully reduced form. Ideally we would like to use global information about the term rewriting system to influence the order in which the matching algorithm generates substitutions. Again this seems difficult.

Acknowledgements

I thank Manuel Clavel, Patrick Lincoln and José Meseguer for many useful discussions on the topics in this paper.

References

[1] L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development, 4th International J oint Conference CAAP/FASE*, pages 61–74. Springer-Verlag, 1993.

[2] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3:203–216, 1987.

[3] J. Bergstra and J. Tucker. A characterization of computable data types by means of a finite equational specification method. In *Automata, Languages and Programming, Seventh Colloquium*, Lecture Notes in Computer Science 81, pages 76–90. Springer-Verlag, 1980.

[4] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. This volume.

[5] S. Eker. Associative matching for linear terms. Technical Report CS-R9224, Center for Mathematics and Computer Science, Amsterdam, July 1992.

[6] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[7] J. Goguen and J. Meseguer. Order sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[8] 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.

[9] J.-M. Hullot. *Compilation de Formes Canoniques dans les Theories Equationnelles.* PhD thesis, University de Paris Sud, Orsay, France, 1980.

[10] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

[11] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[12] T. Nipkow. Combining matching algorithms: The regular case. *Journal of Symbolic Computation*, 12:633–653, 1991.

[13] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In *Automata, Languages and Programming, 19th Colloquium*, Lecture Notes in Computer Science 623, pages 247–260. Springer-Verlag, 1992.