1992

S.M. Eker

Associative matching for linear terms

Computer Science/Department of Software Technology Report CS-R9224 July

CWI is het Centrum voor Wiskunde en Informatica van de Stichting Mathematisch Centrum
CWI is the Centre for Mathematics and Computer Science of the Mathematical Centre Foundation

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Associative Matching for Linear Terms

S. M. Eker[0]

CWI, P.O.Box 4079, 1009 AB Amsterdam.

Abstract

Associative matching is a feature supported by current algebraic specification languages. Deciding the existence of a match modulo associativity is known to be NP-complete in general. We show that for a linear pattern term p and a subject term s the associative matching problem can be decided in $O(|p|.|s|)$ time. Our main result is a new algorithm that for linear p computes the complete set $\overline{M}$ of flattened matching substitutions in $O(|p|.|s| + |\overline{M}|.|Var(p)|)$ time.

Key Words & Phrases: List Matching, Tree Pattern Matching, Congruence Class Rewriting.
1991 Mathematics Subject Classification: 68Q20 [**Theory of Computing**]: Nonnumerical Algorithms; 68Q42 [**Theory of Computing**]: Rewriting Systems;
1991 CR Categories: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems.
Note: This work was done while the author was an ERCIM Research Fellow.

1 Introduction

Associative matching (or a restricted form called list matching) is a feature supported by current algebraic specification languages [5, 4, 3, 2]. The problem of determining whether a match exists modulo associativity is known to be NP-complete [1]. The related problems of associative-commutative matching [6, 9] and unification [13, 10] have been more often considered in the literature. They are also known to be NP-complete [1, 7]; however for a linear pattern term p and a subject term s the existence of a match modulo associativity and commutativity can be decided in $O(|p|.|s|^3)$ time [1]. Associative unification has a extensive literature (for example [12]) although this appears to be a much harder problem (in terms of complexity). Hence it seems unlikely that a useful matching algorithm could be derived from this work.

We show that for a linear pattern term p and a subject term s the existence of a match modulo associativity can be decided in $O(|p|.|s|)$ time. In practice, knowing that a match exists is insufficient. In term rewriting system implementations which handle conditional equations by backtracking it is in general necessary to compute a complete set of matching substitutions. In practice where terms are stored in 'flattened' form, it is the complete set of flattened substitutions that is most useful. We give a naive algorithm for this task and show that there exists a family of associative matching problems with linear pattern terms on which the naive algorithm takes exponential time to discover that no matching substitutions exist. Our main result is a new algorithm for building a graph based data structure in $O(|p|.|s|)$ time and space which encodes the complete set of flattened matching substitutions and from which each flattened matching substitution can be extracted in $O(|Var(p)|)$ time (assuming they are extracted in a particular order and that pointers are used to avoid copying subterms). We also discuss how this algorithm could be applied to the general associative matching problem, however in this case the existence of a match cannot be easily determined from the graph based data structure and the complexity result for extracting flattened matching substitutions is lost.

[0]From 15th July 1992 the author's address will be INRIA-LORRAINE, Campus Scientifique, 615 rue du Jardin Botanique - BP 101, F-54602 VILLERS-LES-NANCY CEDEX, France.

Report CS-R9224
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

2 Basic definitions

Let $\Sigma = \bigcup_{n \geq 0} \Sigma_n$ be a finite signature where each Σ_n is empty or contains n-ary function symbols and let $X = \{x_1, x_2, \ldots\}$ be a countable set of variables. We denote the set of Σ-terms (possibly) containing variables from X by $T_\Sigma(X)$. A position Γ within a term t is a sequence of positive integers which describes a path starting at the outermost function symbol of t to some subterm, denoted by $t|_\Gamma$, within t. For example if $t = f(g(a, x_1), c, f(x_2, b))$ then $t|_1 = g(a, x)$ and $t|_{3.2} = b$.

A term t is *linear* if no variable occurs in t more than once. The set of all variables occurring in a term t is denoted by $Var(t)$. The size of t, denoted by $|t|$, is the total number of occurrences of function symbols and variables in t.

A substitution is a map $\sigma : X \to T_\Sigma(X)$ such that all but finitely many variables are mapped to themselves. The extension of σ to $T_\Sigma(X)$ (in the usual way), applied to $t \in T_\Sigma(X)$ is denoted by $t\sigma$. A term $p \in T_\Sigma(X)$ is said to match a term $s \in T_\Sigma(X)$ iff there exists a substitution such that $p\sigma = s$. We call p the pattern, s the subject and σ a matching substitution.

Let E be a set of Σ-equations. Two terms $t_1, t_2 \in T_\Sigma(X)$ are equal modulo E, denoted $t_1 =_E t_2$ iff $E \vdash t_1 = t_2$ under the usual rules of equational logic. A term $p \in T_\Sigma(X)$ is said to match a term $s \in T_\Sigma(X)$ modulo E iff there exists a substitution σ such that $p\sigma =_E s$.

We are interested matching modulo associativity; i.e. modulo the set of equations

$$A = \{f(x_1, f(x_2, x_3)) = f(f(x_1, x_2), x_3) \mid f \in \Sigma_A\}$$

for some set of associative function symbols $\Sigma_A \subseteq \Sigma_2$.

2.1 Flattened form

When manipulating terms involving associative function symbols it is convenient to represent them in 'flattened' form where nested occurrences (and single unnested occurrences) of an associative function symbol f are replaced by a single variadic function symbol f^*. For example

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

where $f, g \in \Sigma_A$ becomes

$$f^*(a, g^*(a, b), g^*(b, c, c), c).$$

In order to define flattening formally we first define a family of functions $strip_f : T_\Sigma(X) \to (T_\Sigma(X))^+$ where $f \in \Sigma_A$ inductively:

$$strip_f(x) = x \quad \text{for } x \in X$$

$$strip_f(c) = c \quad \text{for } c \in \Sigma_0$$

$$strip_f(g(t_1, \ldots, t_n)) = g(t_1, \ldots, t_n) \quad \text{for } g \in \Sigma_n,\ g \neq f$$

$$strip_f(f(t_1, t_2)) = strip_f(t_1), strip_f(t_2)$$

Intuitively $strip_f$ strips off all outermost occurrences of f to leave a sequence of terms whose outmost function symbols are not f. For convenience we define maximally flattening over nonempty sequences of terms.

$$\overline{x} = x \quad \text{for } x \in X$$

$$\overline{c} = c \quad \text{for } c \in \Sigma_0$$

$$\overline{f(t_1, \ldots, t_n)} = f(\overline{t_1}, \ldots, \overline{t_n}) \quad \text{for } f \in \Sigma_n,\ f \notin \Sigma_A$$

$$\overline{f(t_1, t_2)} = f^*(\overline{strip_f(f(t_1, t_2))})$$

$$\overline{t_1, \ldots, t_n} = \overline{t_1}, \ldots, \overline{t_n}$$

We can extend the operation of maximal flattening to terms that might be partly flattened by the following rule:

$$\overline{f^*(t_1, \ldots, t_n)} = f^*(l_1, \ldots, l_n) \text{ where } l_i = \begin{cases} u_1, \ldots, u_m & \text{if } \overline{t_i} = f^*(u_1, \ldots, u_m) \\ \overline{t_i} & \text{otherwise} \end{cases}$$

We define

$$\overline{T_\Sigma(X)} = \{\bar{t} \mid t \in T_\Sigma(X)\}.$$

The notions of position, linearity, $Var(t)$ and $|t|$ carry over to flattened terms in the obvious way.

Lemma 1 *A term $t \in T_\Sigma(X)$ can be maximally flattened in $O(|t|)$ time.*

Lemma 2 *For all $t \in T_\Sigma(X)$, $2|t| \geq 2|\bar{t}| \geq |t|$.*

Lemma 3 *Suppose $t, s \in T_\Sigma(X)$, then $t =_A s$ iff $\bar{t} = \bar{s}$.*

Lemma 4 *For $t = f(t', t'') \in T_\Sigma(X)$ with $f \in \Sigma_A$ and $\sigma : X \to T_\Sigma(X)$*

$$strip_f(t) = t_1, \ldots, t_n \Rightarrow \overline{t\sigma} = f^*(\overline{strip_f(t_1\sigma)}, \ldots, \overline{strip_f(t_n\sigma)})$$

Intuitively, Lemma 1 will allow us to forget about the cost of flattening terms when discussing the time complexity of our matching algorithms because the worst case complexity of matching will always be at least as great as that of flattening the pattern and subject terms. Lemma 2 allows us to ignore the change of size of a term due to flattening in complexity arguments since it will never be more than a factor of two. Lemma 3 is the whole point of flattening terms—equality modulo associativity becomes syntactic equality. Lemma 4 is a technical lemma which we will need later.

2.2 Block form

A basic idea which we will use in all the algorithms that we describe is that of splitting the argument list of a variadic associative function symbol into variable and nonvariable blocks. Consider a sequence of terms $B = t_1, \ldots, t_n$ where each term is in flattened form or (to be defined later) block form. The length of B, denoted by $l(B)$, is n. We say B is a *variable block* if each $t_i \in X$, and a *nonvariable block* if each $t_i \notin X$,. Clearly any sequence $t_1, \ldots, t_n$ of terms in flattened or block form can be split into an unique shortest sequence of alternating variable and nonvariable blocks $B_1, \ldots, B_q$. Given a term $t \in \overline{T_\Sigma(X)}$ we use $\hat{t}$ to denote the *block form* of t in which the argument list of each variadic function symbol occurring in t has been split into its unique shortest sequence of blocks. For example

$$f^*(a, x_1, x_2, g^*(a, x_3), g^*(a, b), x_4)$$

becomes

$$f^*([a], [x_1, x_2], [g^*([a], [x_3]), g^*([a, b])], [x_4])$$

where each block is delimited by (square) brackets. We will also use the notation $\hat{t}$ where $t \in T_\Sigma(X)$ to denote the result of first flattening t and then converting $\bar{t}$ to block form. We define

$$\widehat{T_\Sigma(X)} = \{\hat{t} \mid t \in T_\Sigma(X)\}.$$

The notion of linearity carries over to terms in block form in the obvious way. A block (variable or nonvariable) is linear if it occurs in the block form of any linear term.

Lemma 5 *A term $t \in \overline{T_\Sigma(X)}$ can be converted into block form in $O(|t|)$ time.*

Notice that converting a term to block form does not change the number of occurrences of function symbols and variables in the term. Although some extra storage maybe required to record the limits of each block this extra storage will be at most linear in the size of the original term and for complexity purposes we will take $|\hat{t}|$ to be $|t|$.

From a theoretical standpoint our notion of blocks is a rather inelegant one. However it turns out that from an algorithmic point of view associative matching can be reduced to searching for ways of partitioning the argument lists of variadic function symbols in the subject amongst the blocks in the arguments lists of corresponding variadic function symbols in the pattern.

Given a linear term $t \in \widehat{T_\Sigma(X)}$, we define $NAV(t)$ to be the set of all variables occurring in t as an argument to a nonassociative function symbol and $VB(t)$ to be the set of all variable blocks occurring in t. For $B \in VB(t)$ we use the notation f_B^* to denote the function symbol in whose argument list B lies when when it occurs in $\hat{t}$.

Definition 1 *A* block substitution ϕ *for* $t \in \widehat{T_\Sigma(X)}$ *is a pair of maps*

$$\phi = \langle \phi_V : NAV(t) \to \overline{T_\Sigma(X)}, \phi_B : VB(t) \to (\overline{T_\Sigma(X)})^+ \rangle$$

such that for all $B \in VB(t)$, $l(\phi_B(B)) \geq l(B)$.

Definition 2 *The application of* ϕ *to subterms of* t *and blocks occurring within* t *are defined by induction:*

$$x\phi = \phi_V(x) \quad \text{for } x \in X.$$

$$c\phi = c \quad \text{for } c \in \Sigma_0.$$

$$(f(t_1, \ldots, t_n))\phi = f(t_1\phi, \ldots, t_n\phi) \quad \text{for } f \in \Sigma_n, f \notin \Sigma_A.$$

$$(f^*(B_1, \ldots, B_q))\phi = \overline{f^*(B_1\phi, \ldots, B_q\phi)} \quad \text{for } f \in \Sigma_A.$$

$$[t_1, \ldots, t_n]\phi = t_1\phi, \ldots, t_n\phi \quad \text{where } t_i \in (\widehat{T_\Sigma(X)} - X) \text{ for } i \in \{1, \ldots, n\}.$$

$$B\phi = \phi_B(B) \quad \text{for } B \in VB(t).$$

The intuition here is that a block substitution ϕ assigns to each variable block B a sequence of flattened terms $\phi_B(B)$ at least as long as B itself so that later this sequence can be split up amongst the variables in B. Each variable x that occurs as an argument of a nonassociative function symbol is assigned a single flattened term $\phi_V(x)$.

Definition 3 *A linear term* $p \in \widehat{T_\Sigma(X)}$ *block-matches a term* $s \in \overline{T_\Sigma(X)}$ *iff there exists a block substitution* ϕ *for* p *such that* $p\phi = s$.

Definition 4 *A block* $B = [p_1, \ldots, p_n]$ *occurring in a linear term* $p \in \overline{T_\Sigma(X)}$ *block-matches a sequence* $S = s_1, \ldots, s_m$ *iff there exists a block substitution* ϕ *for* p *such that* $B\phi = S$.

Theorem 1 *For all* $p, s \in T_\Sigma(X)$ *with* p *linear,* p *matches* s *modulo associativity iff* $\hat{p}$ *block-matches* $\bar{s}$.

Proof: This is a direct consequence of Theorems 5 and 6 (see §4). □

Theorem 2 *Consider* $p \in \widehat{T_\Sigma(X)}$ *and* $s \in \overline{T_\Sigma(X)}$ *with* p *linear. Then the following six statements hold.*

1. *If* $p = x$ *for* $x \in X$ *then* p *block-matches* s.
2. *If* $p = c$ *for* $c \in \Sigma_0$ *then* p *block-matches* s *iff* $s = c$.
3. *If* $p = f(p_1, \ldots, p_n)$ *for* $f \in \Sigma_n, f \notin \Sigma_A$ *then* p *block-matches* s *iff* $s = f(s_1, \ldots, s_n)$ *and* p_i *block-matches* s_i *for* $i \in \{1, \ldots, n\}$.
4. *If* $p = f^*(B_1, \ldots, B_q)$ *for* $f \in \Sigma_A$ *then* p *block-matches* s *iff* $s = f^*(s_1, \ldots, s_m)$ *and* $s_1, \ldots, s_m$ *can be partitioned in to subsequences* $S_1, \ldots, S_q$ *such that* B_i *block-matches* S_i *for* $i \in \{1, \ldots, q\}$.
5. *Let* B *be any variable block occurring in* p *and let* S *be any sequence* $s_1, \ldots, s_m \in (\overline{T_\Sigma(X)})^+$. *Then* B *block-matches* S *iff* $l(B) \leq m$.
6. *Let* $B = p_1, \ldots, p_n$ *be any nonvariable block occurring in* p *and let* S *be any sequence* $s_1, \ldots, s_m \in (\overline{T_\Sigma(X)})^+$. *Then* B *block-matches* S *iff* $n = m$ *and for* $i \in \{1, \ldots, n\}$, p_i *block-matches* s_i.

Proof: [Sketch] Cases 1, 2 and 5 follow trivially from the above definitions. The forwards argument in cases 3, 4 and 6 is to assume that p block matches s and hence there exists ϕ such that $p\phi = s$. Then new block substitutions are constructed for each subterm (block) of p by restricting ϕ_V and ϕ_B to the nonassociative variables and variable blocks occurring in each subterm (block). By substituting in the definition of block matching these new block substitutions can be seen to match the subterms (blocks) of p to the subterms (subsequences) of s. The backwards argument is the reverse of this construction with the linearity of p allowing us to combine block substitutions which match the subterms (blocks) of p to the subterms (subsequences) of s to obtain a new block substitution which matches p to s. □

The intuition behind these two theorems is that we reduce the problem of associative matching to searching for ways of partitioning sequences of terms in the subject amongst sequences of blocks in the pattern that satisfy certain conditions.

In order to simplify the terminology in the rest of this paper we will say a term $p \in T_\Sigma(X)$ matches a term $s \in T_\Sigma(X)$ to mean p matches s modulo associativity and $p \in T_{\widehat{\Sigma}}(X)$ matches $s \in \widehat{T_\Sigma(X)}$ to mean p block-matches s. We will also say that a block B matches a sequence S when we really mean that B block-matches S.

3 Deciding the existence of a match

We now describe a straight forward recursive algorithm, based on Theorem 2, to decide whether a linear term $p \in T_\Sigma(X)$ matches a term $s \in T_\Sigma(X)$ in the presence of associative function symbols Σ_A. We assume that p is in block form and that s is maximally flattened.

A pseudo code description of the algorithm is given in Figures 1 and 2. We assume the existence of a predefined subroutine NONVAR that takes a block B and returns *true* if B is a nonvariable block and *false* otherwise. In the basis case we either have $p = x$ for some variable $x \in X$ in which case we return *true*; or we have $p = c$ for some constant $c \in \Sigma$ in which case we return *true* if $s = c$ and *false* otherwise.

In the inductive case the outermost function symbol is either associative or nonassociative. The nonassociative case is straight forward. Suppose $p = f(p_1, \ldots, p_n)$ for some $f \in \Sigma_n, f \notin \Sigma_A$. Then if $s = f(s_1, \ldots, s_n)$ and each p_i matches each s_i for $i \in \{1, \ldots, n\}$ we return *true*; otherwise we return *false*.

The case where the outermost function symbol is associative is more complicated. Suppose $p = f^*(B_1, \ldots, B_q)$ for some $f \in \Sigma_A$. Now unless $s = f^*(s_1, \ldots, s_m)$ we return *false* immediately. Otherwise we have to check whether $s_1, \ldots, s_m$ can be split into subsequences $S_1, \ldots, S_q$ such that for $i \in \{1, \ldots, q\}$, B_i matches S_i. If m is less than the total length of $B_1, \ldots, B_q$ then such a splitting is clearly impossible and we return *false* immediately.

We now consider the first and last blocks, B_1 and B_q. Clearly these must match at the left and right ends of $s_1, \ldots, s_m$ respectively. If they are nonvariable blocks we know how large any matching subject subsequences would be so we can check to see if they match and then disregard them. We also take care of the special case where the argument list of the pattern term consists of a single nonvariable block. The (algorithm) variables *first_var* and *last_var* are set to the indices of first and last variable blocks after any leading or trailing nonvariable blocks have been matched and disregarded. The (algorithm) variables *first_sub* and *last_sub* are set to the indices of the first and last unmatched subject terms from $s_1, \ldots, s_m$.

Now if *last_var* = *first_var* then we have one variable block left and no nonvariable blocks left[1] and we return *true*[2]. Otherwise we have at least two variable blocks and at least one nonvariable block unmatched. We call the subroutine SLIDE to search through the possibilities for matching the remaining blocks against the remaining subject terms.

3.1 The partitioning subproblem

We now consider the subroutine SLIDE. It has to determine whether $s_1, \ldots, s_m$ can be split into subsequences $S_1, \ldots, S_n$ such that for $i \in \{1, \ldots, n\}$, B_i matches S_i. We already know (from the context in which SLIDE is called) that B_i is a variable block for i odd and a nonvariable block for i even, that $n \geq 3$ is odd and that $m \geq \sum_{i=1}^{n} l(B_i)$. Since a variable block B will match a subsequence of subject terms of length greater or equal to $l(B)$ the problem reduces to finding matching subsequences of subject terms for the nonvariable blocks which respect this length constraint on the interleaved variable blocks. Formally we want to find for each $i \in \{2, 4, , \ldots, n-3, n-1\}$ an index α_i such that:

1. For $i \in \{2, 4, , \ldots, n-3, n-1\}$ the nonvariable block B_i matches the sequence $s_{\alpha_i}, \ldots, s_{\alpha_i + l(B_i) - 1}$.

[1] Remember variable and nonvariable blocks must alternate.

[2] Since any disregarded nonvariable blocks will have matched subject subsequences of the same length, we know that the number of remaining subject terms will be greater or equal to the length of the remaining nonvariable block.

```
MATCH(x, s) where x ∈ X is
    return(true).

MATCH(c, s) where c ∈ Σ_0 is
    if s = c then return(true) else return(false) fi.

MATCH(f(p_1, ..., p_n), s) where f ∈ Σ_n, f ∉ Σ_A is
    if s = f(s_1, ..., s_n) then
        return(MATCH_SEQ([p_1, ..., p_n], [s_1, ..., s_n]))
    else
        return(false)
    fi.

MATCH(f*(B_1, ..., B_q), s) where f ∈ Σ_A is
    if s = f*(s_1, ..., s_m) and m ≥ Σ_{i=1}^{q} l(B_i) then
        if NONVAR(B_1) then
            if q = 1 then return(MATCH_SEQ(B_1, [s_1, ..., s_m])) fi;
            if MATCH_SEQ(B_1, [s_1, ..., s_{l(B_1)}]) = false then return(false) fi
            first_var := 2; first_sub := 1 + l(B_1)
        else
            first_var := 1; first_sub := 1
        fi;
        if NONVAR(B_q) then
            if MATCH_SEQ(B_1, [s_{m-l(B_q)+1}, ..., s_m]) = false then return(false) fi
            last_var := q - 1; last_sub := m - l(B_q)
        else
            last_var := q; last_sub := m
        fi;
        if first_var = last_var then
            return(true)
        else
            return(SLIDE([B_{first_var}, ..., B_{last_var}], [s_{first_sub}, ..., s_{last_sub}]))
        fi
    else
        return(false)
    fi.
```

Figure 1: Algorithm 1 (main routine).

2. $\alpha_2 > l(B_1)$.
3. $\alpha_{n-1} + l(B_{n-1}) + l(B_n) \leq m + 1$.
4. For $i \in \{2, 4, , \ldots, n-5, n-3\}$, $\alpha_i + l(B_i) + l(B_{i+1}) \leq \alpha_{i+1}$.

We do this by finding such an α_i for each nonvariable block in turn. At the start of the **while** loop, the (algorithm) variable α records the first candidate for α_i that satisfies conditions 2 and 4 and the (algorithm) variable β records the last candidate for α_i that could possibly satisfy conditions 3 and 4 (given that we know the lengths of all the unmatched blocks). If at any stage we fail to find an α_i by incrementing α to β that satisfies condition 1 then we can return *false* immediately. This is because backtracking to find a larger index for an earlier nonvariable block will only narrow the field of candidates for α_i. It is this early failure detection that keeps the time complexity of the search polynomial.

```
SLIDE([B_1, ..., B_n], [s_1, ..., s_m]) is
    α = 1; β := 1 + m − Σ_{i=1}^{n} l(B_i);
    for i := 2 to n − 1 step 2 do
        α := α + l(B_{i−1}); β := β + l(B_{i−1});
        while MATCH_SEQ(B_i, [s_α, ..., s_{α+l(B_i)−1}]) = false do
            α := α + 1;
            if α > β then return(false) fi
        od;
        α := α + l(B_i); β := β + l(B_i)
    od;
    return(true).

MATCH_SEQ([p_1, ..., p_n], [s_1, ..., s_m]) is
    if n ≠ m then return(false) fi;
    for i := 1 to n do
        if MATCH(p_i, s_i) = false then return(false) fi
    od;
    return(true).
```

Figure 2: Algorithm 1 (subroutines).

3.2 Implementation and complexity

We first note that in order to flatten a term t in $O(|t|)$ time we need to be able to concatenate the argument lists of associative function symbols in constant time. An easy way of doing this is to store them as linked lists where pointers to both the head and tail are maintained. Notice that in Algorithm 1 we access the elements of an argument list by index. To allow each access to be done in constant time we can copy each argument list into an array at the outset.

Theorem 3 *Algorithm 1 running on pattern p and subject s takes $O(|p|.|s|)$ time.*

Proof: Let the time taken by Algorithm 1 running on pattern p and subject s be $cost(p, s)$. We first obtain an expression to bound this cost in each of the four cases.

Case 1: $p \in X$
Clearly $cost(p, s) = K_1$ for some constant K_1.

Case 2: $p \in \Sigma_0$
Clearly $cost(p, s) = K_2$ for some constant K_2.

Case 3: $p = f(p_1, \ldots, p_n)$ for $f \in \Sigma_n$, $f \notin \Sigma A$
Then $cost(p, s)$ is maximised when $s = f(s_1, \ldots, s_n)$ and the algorithm could call itself (via MATCH_SEQ) on each pair (p_i, s_i) for $i \in \{1, \ldots, n\}$. We have

$$cost(p, s) \leq K_3 n + \sum_{i=1}^{n} cost(p_i, s_i)$$

for some constant K_3.

Case 4: $p = f^*(B_1, \ldots, B_q)$ for $f \in \Sigma_A$
Then $cost(p, s)$ is maximised when $s = f^*(s_1, \ldots, s_m)$ and $m \geq \sum_{i=1}^{q} l(B_i)$. Now it is clear that each nonvariable block B_k is checked for a match against each subsequence of $s_1, \ldots, s_m$ at most once. Suppose in the maximally flattened form of p the argument list of f^* is $p_1, \ldots, p_n$. Then MATCH is called

(indirectly via MATCH_SEQ and possibly SLIDE) at most once on each pair of terms p_i, s_j. We have

$$cost(p, s) \leq K_4 nm + \sum_{i=1}^{n} \sum_{j=1}^{m} cost(p_i, s_j)$$

for some constant K_4. We now prove an explicit bound on $cost(p, s)$ by induction on the structure of p. Our induction hypothesis is that $cost(p, s) \leq 2K.|p|.|s| - K.|p|$ for some constant K such that $K > K_i$, $i \in \{1, \ldots, 4\}$. The basis cases are trivial. We consider the two induction cases. When $p = f(p_1, \ldots, p_n)$ for $f \in \Sigma_n$, $f \notin \Sigma_A$ we have

$$\begin{aligned}
cost(p, s) &\leq K_3 n + \sum_{i=1}^{n} cost(p_i, s_i) \\
&\leq K_3 n + \sum_{i=1}^{n} (2K.|p_i|.|s_i| - K.|p_i|) \quad \text{(by induction hypothesis)} \\
&\leq K_3 n + 2K(|p| - 1)(|s| - 1) - K(|p| - 1) \\
&\leq 2K.|p|.|s| - K.|p| + K_3 n - 2K.|p| - 2K.|s| + 2K + K \\
&\leq 2K.|p|.|s| - K.|p| \quad \text{(since } |p| > n \text{ and } |s| > 1)
\end{aligned}$$

When $p = f^*(B_1, \ldots, B_q)$ for $f \in \Sigma_A$ we have

$$\begin{aligned}
cost(p, s) &\leq K_4 nm + \sum_{i=1}^{n} \sum_{j=1}^{m} cost(p_i, s_j) \\
&\leq K_4 nm + \sum_{i=1}^{n} \sum_{j=1}^{m} (2K.|p_i|.|s_j| - K.|p_i|) \quad \text{(by induction hypothesis)} \\
&\leq K_4 nm + 2K(|p| - 1)(|s| - 1) + 2K - Km(|p| - 1) \\
&\leq 2K.|p|.|s| - K.|p| + K_4 nm - 2K.|p| - 2K.|s| + 2K - K(m - 1)|p| + Km \\
&\leq 2K.|p|.|s| - K.|p| + K_4 n - 2K.|p| - 2K.|s| + 2K + Km \\
&\leq 2K.|p|.|s| - K.|p| \quad \text{(since } |p| > n \text{ and } |s| > m \geq 1)
\end{aligned}$$

This completes the induction step and $cost(p, s) =. O(|p|.|s|)$ follows immediately. □

4 Finding a complete set of matching substitutions

For most applications, just determining whether p matches s modulo associativity is insufficient. It is necessary to compute a complete set of matching substitutions.

Definition 5 *A set M of associative matching substitutions for p and s is complete iff for every σ such that $p\sigma =_A s$ there exists $\sigma' \in M$ such that for all $x \in Var(p)$, $\sigma(x) =_A \sigma'(x)$.*

For each substitution $\sigma : X \to T_\Sigma(X)$ such that $p\sigma =_A s$ we define a flattened substitution $\overline{\sigma} : X \to \overline{T_\Sigma(X)}$ as follows:

$$\overline{\sigma}(x) = \begin{cases} \overline{\sigma(x)} & \text{if } x \in Var(p) \\ x & \text{if } x \notin Var(p) \end{cases}$$

Intuitively flattening a substitution in this way removes redundant information concerning the nesting of associative function symbols and the value of the substitution on variables not involved in the matching process. We define

$$\overline{M} = \{\overline{\sigma} \mid \sigma : X \to T_\Sigma(X), p\sigma =_A s\}.$$

Since most applications which use associative matching will keep all terms in flattened form it is this $\overline{M}$ (which is equivalent to the flattening of any complete set of matching substitutions) which we want to compute in practice.

Now suppose p is linear. Consider the set $\widehat{M}$ of all block substitutions ϕ for $\hat{p}$ such that $\hat{p}\phi = \overline{s}$. Given some $\phi \in \widehat{M}$ we present a (nondeterministic) scheme for constructing a flattened substitution, $\psi : X \rightarrow \overline{T_\Sigma(X)}$ as follows.

1. For each variable block $B = [x_1, \ldots, x_{l(B)}]$ we partition $\phi_B(B)$ into $l(B)$ nonempty subsequences $S_1, \ldots, S_{l(B)}$. Now for $i \in \{1, \ldots, l(B)\}$ we put

$$\psi(x_i) = \begin{cases} S_i & \text{if } l(S_i) = 1 \\ f^*(S_i) & \text{if } l(S_i) \geq 2 \end{cases}$$

2. For each $x \in NAV(\hat{p})$ we put

$$\psi(x_i) = \phi_V(x_i)$$

3. For each $x \in (X - Var(p))$ we put

$$\psi(x_i) = x$$

Now for each variable block B we have

$$\begin{pmatrix} l(\phi_B(B)) - 1 \\ l(B) - 1 \end{pmatrix}$$

ways of splitting $\phi_B(B)$ into $l(B)$ nonempty subsequences. Since this choice is independent for each $B \in VB(\hat{p})$ we can construct a family M_ϕ of

$$\prod_{B \in VB(t)} \begin{pmatrix} l(\phi_B(B)) - 1 \\ l(B) - 1 \end{pmatrix}$$

substitutions for each block substitution ϕ using this scheme.

Theorem 4 *For $\phi, \phi' \in \widehat{M}$, if $\phi \neq \phi'$ then $M_{\phi'}$ and $M_{\phi'}$ are disjoint.*

Proof: Suppose $\phi_V(x) \neq \phi'_V(x)$ for some $x \in NAV(\hat{p})$. Then clearly for any $\psi \in M_\phi$ and $\psi' \in M_{\phi'}$ we have $\psi(x) \neq \psi'(x)$. Alternatively suppose $\phi_B(B) \neq \phi'_B(B)$ for some $B = [x_1, \ldots, x_{l(B)}] \in VB(\hat{p})$. Now when we split $\phi_B(B)$ up into $S_1, \ldots, S_{l(B)}$ it follows that for at least one $i \in \{1, \ldots, l(B)\}$ we have $S_i \neq S'_i$ and hence for any $\psi \in M_\phi$ and $\psi' \in M_{\phi'}$ we have $\psi(x_i) \neq \psi'(x_i)$. □

Theorem 5 *Let $p, s \in T_\Sigma(X)$ with p linear, let ϕ be a block substitution for $\hat{p}$ such that $\hat{p}\phi = \overline{s}$ and let $\psi \in M_\phi$. Then there exists $\sigma : X \rightarrow T_\Sigma(X)$ such that $p\sigma =_A s$ and $\overline{\sigma} = \psi$.*

Proof: For each $t \in \overline{T_\Sigma(X)}$ we define $\tilde{t} \in T_\Sigma(X)$ to be the term which flattens to t and in which all occurrences of associative functions symbols are arranged in 'left associative' form. We define σ by

$$\sigma(x) = \widetilde{\psi(x)}$$

Now trivially $\overline{\sigma} = \psi$. To see that $p\sigma =_A s$ we prove that $\overline{p\sigma} = \hat{p}\phi$ by induction on the structure of p. There are two basis cases. If $p \in X$ then

$$\overline{p\sigma} = \overline{\sigma(p)} = \psi(p) = \phi_V(p) = \hat{p}\phi$$

otherwise if $p \in \Sigma_0$ then

$$\overline{p\sigma} = p = \hat{p}\phi$$

For the induction step we also consider two cases. If $p = f(p_1, \ldots, p_n)$ where $f \in \Sigma_n$, $f \notin \Sigma_A$ then

$$\overline{p\sigma} = \overline{f(p_1\sigma, \ldots, p_n\sigma)} = f(\overline{p_1\sigma}, \ldots, \overline{p_n\sigma}) = f(\widehat{p_1}\phi, \ldots, \widehat{p_n}\phi) = \hat{p}\phi$$

using the induction hypothesis. Finally if $p = f(t_1, t_2)$ for $f \in \Sigma_A$ we proceed as follows. Let $strip_f(p) = p_1, \ldots, p_n$. Then

$$\overline{p} = f^*(\overline{p_1}, \ldots, \overline{p_n})$$

and

$$\hat{p} = f^*(B_1, \ldots, B_q) \quad \text{where } B_i = [\widehat{p_{\alpha(i)}}, \ldots, \widehat{p_{\omega(i)}}].$$

Now

$$\hat{p}\phi = f^*(B_1\phi, \ldots, B_q\phi)$$

and

$$\overline{p\sigma} = f^*(\overline{strip_f(p_1\sigma)}, \ldots, \overline{strip_f(p_n\sigma)})$$

by Lemma 4 so we want to show for $i \in \{1, \ldots, q\}$ that

$$\overline{strip_f(p_{\alpha(i)}\sigma)}, \ldots, \overline{strip_f(p_{\omega(i)}\sigma)} = B_i\phi$$

We consider two cases.

Case 1: B_i is a variable block
Then for $k \in \{\alpha(i), \ldots, \omega(i)\}$, p_k is a variable and we have

$$\overline{strip_f(\sigma(p_k))} = \begin{cases} S & \text{if } \psi(p_k) = f^*(S) \\ \psi(p_k) & \text{otherwise} \end{cases}$$

so

$$\overline{strip_f(\sigma(p_{\alpha(i)}))}, \ldots, \overline{strip_f(\sigma(p_{\omega(i)}))} = \phi_B(B_i)$$

since we are effectively reversing step 1 in the above scheme for constructing a flattned substitution ψ from ϕ.

Case 2: B_i is a nonvariable block
Then

$$B_i\phi = \widehat{p_{\alpha(i)}}\phi, \ldots, \widehat{p_{\omega(i)}}\phi = \overline{p_{\alpha(i)}\sigma}, \ldots, \overline{p_{\omega(i)}\sigma}$$

using the induction hypothesis and

$$\overline{strip_f(p_{\alpha(i)}\sigma)}, \ldots, \overline{strip_f(p_{\omega(i)}\sigma)} = \overline{p_{\alpha(i)}\sigma}, \ldots, \overline{p_{\omega(i)}\sigma}$$

since each p_k for $k \in \{\alpha(i), \ldots, \omega(i)\}$ is a nonvariable term whose outmost function symbol is not f and thus $p_k\sigma$ is a nonvariable term whose outmost function symbol is not f. □

Theorem 6 *Let $p, s \in T_\Sigma(X)$ with p linear and let $\sigma : X \to T_\Sigma(X)$ such that $p\sigma =_A s$. Then there exists a block substitution ϕ for $\hat{p}$ such that $\hat{p}\phi = \overline{s}$ and $\overline{\sigma} \in M_\phi$.*

Proof: We construct ϕ as follows. For $x \in NAV(\hat{p})$,

$$\phi_V(x) = \overline{\sigma(x)}.$$

For $B = [x_1, \ldots, x_{l(B)}] \in VB(\hat{p})$

$$\phi_B(x) = \overline{strip_{f_B}(\sigma(x_1))}, \ldots, \overline{strip_{f_B}(\sigma(x_{l(B)}))}$$

Now by applying the above scheme for constructing flattened substitutions from ϕ it is clear that $\overline{\sigma} \in M_\phi$. To see that $\hat{p}\phi = \overline{s}$ we prove $\hat{p}\phi = \overline{p\sigma}$ by induction on the structure of p. There are two basis cases. If $p \in X$ then

$$\hat{p}\phi = \phi_V(p) = \overline{\sigma(p)} = \overline{p\sigma}$$

otherwise if $p \in \Sigma_0$ then

$$\hat{p}\phi = p = \overline{p\sigma}$$

For the induction step we also consider two cases. If $p = f(p_1, \ldots, p_n)$ where $f \in \Sigma_n$, $f \notin \Sigma_A$ then

$$\overline{p\sigma} = \overline{f(p_1\sigma, \ldots, p_n\sigma)} = f(\overline{p_1\sigma}, \ldots, \overline{p_n\sigma}) = f(\widehat{p_1}\phi, \ldots, \widehat{p_n}\phi) = \hat{p}\phi$$

using the induction hypothesis. Finally if $p = f(t_1, t_2)$ for $f \in \Sigma_A$ we proceed as follows. As is the previous proof we have

$$\hat{p}\phi = f^*(B_1\phi, \ldots, B_q\phi)$$

and

$$\overline{p\sigma} = f^*(\overline{strip_f(p_1\sigma)}, \ldots, \overline{strip_f(p_n\sigma)})$$

so we want to show for $i \in \{1, \ldots, q\}$ that

$$\overline{strip_f(p_{\alpha(i)}\sigma)}, \ldots, \overline{strip_f(p_{\omega(i)}\sigma)} = B_i\phi$$

We consider two cases.

Case 1: B_i is a variable block
Then for $k \in \{\alpha(i), \ldots, \omega(i)\}$ we have

$$\phi_B(B) = \overline{strip_f(\sigma(p_{\alpha(i)}))}, \ldots, \overline{strip_f(\sigma(p_{\omega(i)}))}$$

Case 2: B_i is a nonvariable block
The argument is identical to that of case 2 in the previous theorem. □

Corollary 1 $\overline{M} = \bigcup_{\phi \in \widehat{M}} M_\phi$.

Theorem 7 *Given $\phi \in \widehat{M}$, M_ϕ can be computed in $O(|M_\phi|.|Var(p)|)$ time.*

Proof: The assignment to each variable $x \in NAV(\hat{p})$ is $\phi_V(x)$ for every $\psi \in M_\phi$ and this is computed in constant time by copying a pointer. Thus the time to compute the assignments of all $x \in NAV(\hat{p})$ in all $\psi \in M_\phi$ is $O(|NAV(\hat{p})|)$. An easy way of computing the assignments of variables in $Var(p) - NAV(\hat{p})$ for each $\psi \in M_\phi$ is to have a recursive procedure that iteratively computes each partitioning of $\phi_B(B)$ for one block $B \in VB(\hat{p})$ and calls itself to deal with the remaining blocks. Computing each partitioning of $\phi_B(B)$ into $l(B)$ nonempty sequences can be done in $O(l(B))$ time by using pointers and making use of the previous partitioning for that block. Thus computing a partitioning for each block requires at most $O(|Var(p) - NAV(\hat{p})|)$ time. Summing over all flattened matching substitutions we require at most $O(|Var(p) - NAV(\hat{p})|.|M_\phi|)$ time. Thus generating assignments for all $x \in Var(p)$ for all $\psi \in M_\phi$ can be done in $O(|NAV(\hat{p})|) + O(|Var(p) - NAV(\hat{p})|.|M_\phi|) = O(|Var(p)|.|M_\phi|)$ time. □

4.1 A naive algorithm

We now consider the problem of computing $\widehat{M}$ for linear $p \in \widehat{T_\Sigma(X)}$ and $s \in \overline{T_\Sigma(X)}$. A naive algorithm for doing this is given in Figure 3. The idea is to use recursion to do a depth first search for matching block substitutions. Initially the procedure MATCH is called on the set $A = \{\langle p, s\rangle\}$. Intuitively the set A holds all the currently unmatched subject pairs. Each pair may consist of a pattern term p and a subject term s or a sequence of blocks $B_1, \ldots, B_q$ and a sequence of subject terms $s_1, \ldots, s_m$. The current block substitution is built up in global variables ϕ_V, ϕ_B. If A becomes empty then a matching substitution has been found and a subroutine RECORD_MATCH is called to record or otherwise make use of the matching substitution. Otherwise some pair $\langle p, s\rangle$ is chosen and removed from A.

If p is a variable we simply assign s to it in the substitution ϕ and make a recursive call to MATCH to deal with the remaining members of A.

If p is a constant then if $p = s$ we make a recursive call to MATCH to deal with the remaining members of A. Otherwise we have reached a dead end and must backtrack to look for other ways of completing a matching substitution.

If $p = f(p_1, \ldots, p_n)$ then if $s = f(s_1, \ldots, s_n)$ we make a recursive call to deal with the remaining members of A together with $\langle p_1, s_1\rangle, \ldots, \langle p_n, s_n\rangle$. Otherwise we backtrack.

If $p = f^*(B_1, \ldots, B_q)$ then unless $s = f^*(s_1, \ldots, s_m)$ with $m \geq \sum_{i=1}^{q} l(B_i)$ there cannot be a match and we backtrack immediately. Otherwise we make a recursive call to deal with the remaining members of A together with $\langle [B_1, \ldots, B_q], [s_1, \ldots, s_m]\rangle$.

```
global φ_V, φ_B.
MATCH(A) ::=
    if A = ∅ then RECORD_MATCH(φ_V, φ_B); return fi;
    choose ⟨p, s⟩ ∈ A;
    A := A − {⟨p, s⟩};
    if p ∈ X then φ_V(p) = s; MATCH(A); return fi;
    if p ∈ Σ_0 and p = s then MATCH(A); return fi;
    if p = f(p_1, ..., p_n) and s = f(s_1, ..., s_n) then
        MATCH(A ∪ {⟨p_1, s_1⟩, ..., ⟨p_n, s_n⟩}); return
    fi;
    if p = f*(B_1, ..., B_q) and q = f*(s_1, ..., s_m) and m ≥ Σ_{i=1}^{q} l(B_i) then
        MATCH(A ∪ {⟨[B_1, ..., B_q], [s_1, ..., s_m]⟩}); return
    fi
    if p = [B_1, ..., B_q] and q = [s_1, ..., s_m] then
        if q = 1 then
            if NONVAR(B_1) then
                if B_1 = [p_1, ..., p_m] then
                    MATCH(A ∪ {⟨p_1, s_1⟩, ..., ⟨p_m, s_m⟩})
                fi
            else
                φ_B(B_1) := s;
                MATCH(A)
            fi;
            return
        fi;
        if NONVAR(B_1) then
            let B_1 be [p_1, ..., p_{l(B_1)}];
            MATCH(A ∪ {⟨p_1, s_1⟩, ..., ⟨p_{l(B_1)}, s_{l(B_1)}⟩} ∪ {⟨[B_2, ..., B_q], [s_{l(B_1)+1}, ..., s_m]⟩})
        else
            e := m − Σ_{i=2}^{q} l(B_i);
            for i := l(B_1) to e do
                φ_B(B_1) := [s_1, ..., s_i];
                MATCH(A ∪ {⟨[B_2, ..., B_q], [s_{i+1}, ..., s_m]⟩})
            od
        fi
    fi;
    return.
```

Figure 3: Algorithm 2.

The final case is where $p = [B_1, \ldots, B_q]$ and $q = [s_1, \ldots, s_m]$. If $q = 1$ then we have a special case where B_1 must match all of $s_1, \ldots, s_m$ or else this branch of the search will fail. If $q \geq 2$ we just consider B_1. If B_1 is a nonvariable block then it must match the first $l(B_1)$ elements of $s_1, \ldots, s_m$ in order for this branch of the search not to fail. If B_1 is a variable block then this branch of the search splits into sub-branches with one recursive call for each possible initial subsequence of $s_1, \ldots, s_m$ that B_1 might match given that there must be enough unmatched elements of $s_1, \ldots, s_m$ left to match against $B_2, \ldots, B_q$ at a later stage in the search.

For efficient implementation the set A might be replaced by a stack of unmatched pairs with the choose operation always selecting the top pair. Then the local instances of A in each invocation of MATCH could be stored as a single global persistent stack [11].

However there is a serious problem with this backtracking approach to finding all substitutions.

Consider the family of matching problems $\langle P_n, S_n \rangle$ where

$$P_n = f^*(\underbrace{[x_1],[a]}, \underbrace{[x_2],[a]}, \ldots, \underbrace{[x_i],[a]}, \ldots, \underbrace{[x_n],[a]}, \underbrace{[x_{n+1}],[b]})$$

$$S_n = f^*(\overbrace{\underbrace{b,a}, \underbrace{b,a}, \ldots, \underbrace{b,a}}^{2n+1 \text{ pairs}})$$

Now clearly each such problem has no matching substitution and the sizes of P_n and S_n are linear in n so the fact that there is no matching substitution can be discovered in $O(n^2)$ time using Algorithm 1. However if we look at the backtracking Algorithm 2 must do it is clear that there are C_n^{2n} ways of matching the first n pairs of blocks in the argument list of f^* in P_n and all of these will be considered. Therefore a lower bound for the running time of Algorithm 2 on this family of problems is $\Omega(C_n^{2n}) = \Omega(2^n)$.

5 Graph based associative matching algorithm

The backtracking solution is inefficient in two ways. Firstly, as we saw in the last example, there is no early failure detection for branches in the search tree that must eventually fail. Secondly, suppose we are trying to match a sequence of blocks $B_1, \ldots, B_n$ against a subject sequence $s_1, \ldots, s_m$ and that we have found a partitioning of $s_1, \ldots, s_m$ into subsequences $S_1, \ldots, S_n$ such that for all $i \in \{1, \ldots, n\}$, B_i matches S_i. Then when we backtrack to look for some new partitioning that differs at say, S_i, we have to find matches for all B_j, $j > i$ even though much of this work may have already been done before.

Our solution is to modify Algorithm 1 so that instead of returning *true* or *false*, each call to MATCH(p, s) returns either *fail* or the set of all matching block substitutions for p and s. Since this set may be rather large, we will encode it by a pair $\langle S, G \rangle$ where S is a set of variable assignments and G is a Directed Acyclic Graph (DAG). Intuitively S will contain those variable assignments that are identical in all matching block substitutions, while each path between two distinguished nodes *start* and *end* in G will correspond to a substitution, with labels on the edges to keep track of variable assignments. This approach has two advantages; Firstly the DAG can be kept to a size which is polynomial in $|p|$ and $|s|$. Secondly we will build the graph in a way such that the same matching subproblem is never considered twice.[3]

5.1 Matching DAGs in a restricted case

Before explaining the full method, we will first consider a simpler subproblem to illustrate the essential ideas. Recall the subroutine SLIDE from Algorithm 1. Its input is a sequence of blocks $B_1, \ldots, B_n$ and a sequence of subject terms $s_1, \ldots, s_m$. For simplicity we assume that none of the blocks contain associative operators. We want to build a DAG which encodes all of the possible partitionings of $s_1, \ldots, s_m$ into subsequences $S_1, \ldots, S_n$ such that B_i matches S_i for $i \in \{1, \ldots, n\}$.

Now for $k \in \{2, 4, \ldots, n-3, n-1\}$ we know B_k is a nonvariable block. We define

$$Ind_k = \{\alpha \mid B_k \text{ matches } s_\alpha, \ldots, s_{\alpha+l(B_i)-1} \text{ and } \alpha > \textstyle\sum_{i=1}^{k-1} l(B_i) \text{ and } \alpha + \sum_{i=k}^{n} l(B_i) \leq m+1\}$$

Intuitively Ind_k is the set of indices of subject terms in $s_1, \ldots, s_m$ at which the subsequence S_k could start, given that B_k must match S_k and the lengths of the other blocks must be respected. Clearly if for some $k \in \{2, \ldots, n-1\}$, $Ind_k = \emptyset$ then there can be no solution. Otherwise we construct a DAG as follows.

For each set Ind_k we have a set $Node_k$ of nodes with a node labelled α in $Node_k$ for each $\alpha \in Ind_k$. We also have a set $Node_{n+1}$ which contains a single node labelled with $m+1$. The distinguished nodes *start* and *end* are the node in $Node_2$ with least label and the single node in $Node_{n+1}$ respectively.

Each node has potentially two distinct edges leaving it, an *across* edge and a *down* edge. For $k \in \{2, 4, \ldots, n-3, n-1\}$, each node $v \in Node_k$, except the one with greatest label, has an *across* edge

[3] We consider identical subterms occurring at different positions to be distinct in this context.

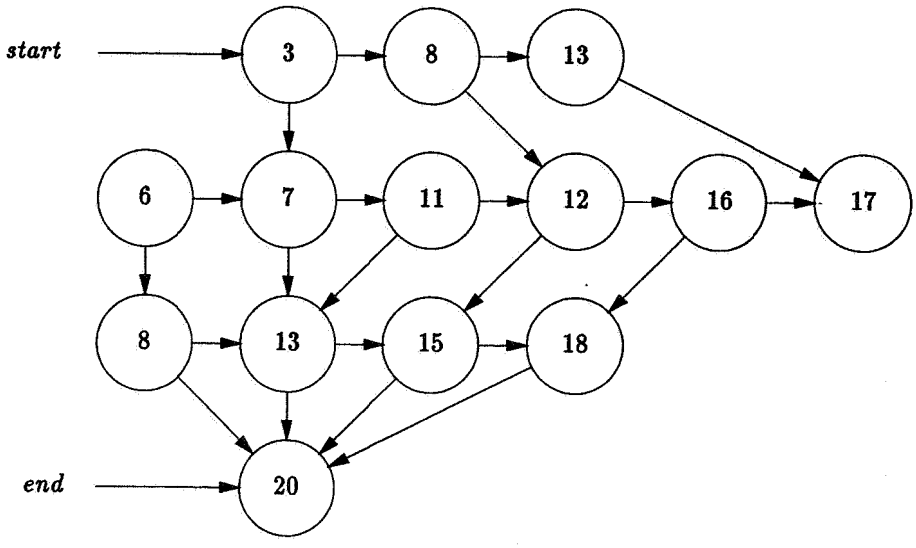

Figure 4: Primitive Matching DAG for Example 1

going to the node in $Node_k$ with next highest label. Each node $v \in Node_k$ also has a *down* edge going to the node $v' \in Node_{k+2}$ with least label such that $label(v') \geq label(v) + l(B_k) + l(B_{k+1})$ if such a node v' exists. We call this structure the *primitive matching DAG* for $B_1, \ldots, B_n$ and $s_1, \ldots, s_m$. We illustrate this idea with an example.

Example 1 *Consider the sequence of blocks* [4]

$$[x_1], [a, b], [x_2, x_3], [c], [x_4], [a], [x_5]$$

and the sequence of subject terms

$$a, a, a, b, b, c, c, a, b, b, c, c, a, b, a, c, c, a, b$$

The sets of indices are:

$$Ind_2 = \{3, 8, 13\}$$

$$Ind_4 = \{6, 7, 11, 12, 16, 17\}$$

$$Ind_5 = \{8, 13, 15, 18\}$$

The primitive matching DAG is shown in Figure 4.

Observe that such a matching DAG effectively encodes all combinations of choices for starting indices α_i for each S_i that meet the four conditions in §3.1 as the set of all paths from the *start* node to the *end* node. The *down* edge from a node $v \in Node_k$ labelled α occurring in a path represents the choice of $S_k = s_\alpha, \ldots, s_{\alpha+l(B_i)-1}$ and restricts the choice for S_{k+2} to subsequences starting at index $\alpha' \geq \alpha + l(B_k) + l(B_{k+1})$.

Notice that some nodes may not lie on any path from *start* to *end*. We define the *reduced matching DAG* for $B_1, \ldots, B_n$ and $s_1, \ldots, s_m$ by eliminating such nodes together with any edges that enter or leave them. The reduced matching DAG for Example 1 is shown in Figure 5.

In order to generate block substitutions from the *start* to *end* paths in a reduced matching graph we introduce labels on the *down* edges. Each *down* edge from a node $v \in Node_k$ has three effects on the block substitution corresponding to a path in which it occurs:

[4] Note that in this example all nonvariable blocks are sequences of constants and all subject terms are constants; this is simply so that an example of sufficient complexity can be illustrated in a reasonable amount of space.

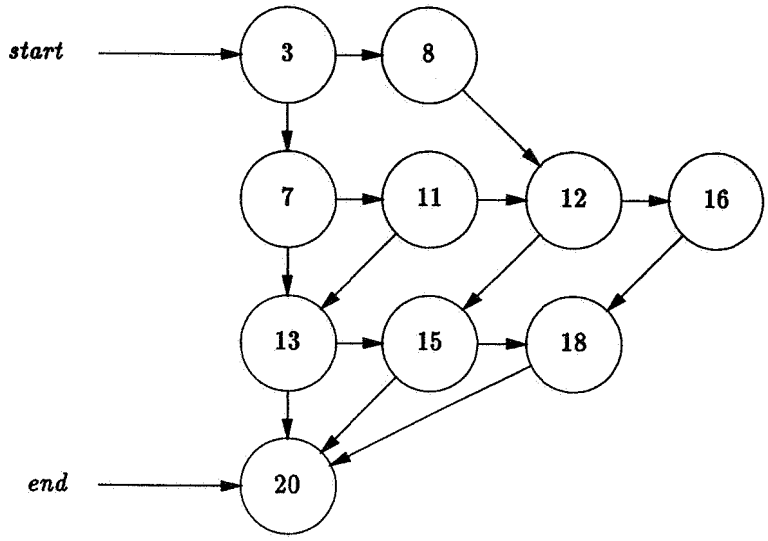

Figure 5: Reduced matching DAG for Example 1

1. It indicates the end of the subsequence S_{k-1} for the previous variable block B_{k-1}.
2. It indicates the substitutions for any variables occurring in the nonvariable block B_k.
3. It indicates the start of the subsequence S_{k+1} for the next variable block B_{k+1}.

Thus we label each *down* edge with a set of triples, where each triple can have one of three forms, which correspond to the three effects:

1. $\langle B_k, \alpha, \rightarrow \rangle$ indicates that S_k starts with s_α.
2. $\langle x_i, t, \leftrightarrow \rangle$ indicates that $\phi_V(x_i) = t$.
3. $\langle B_k, \alpha, \leftarrow \rangle$ indicates that S_k ends with s_α.

The edge between $v \in Node_k$ with label α and $v' \in Node_n$ will be labelled with a set of triples containing $\langle B_{k-1}, \alpha - 1, \leftarrow \rangle$ and $\langle B_{k+1}, \alpha + l(B_k), \rightarrow \rangle$ together with a triple $\langle x_i, t, \leftrightarrow \rangle$ for each variable x_i occurring in B_k where t is the unique[5] term that must be assigned to x_i in any matching block substitution for B_k and $s_\alpha, \ldots, s_{\alpha+l(B_k)-1}$. We call this new structure formed by labelling the *down* edges the *labelled matching DAG* for $B_1, \ldots, B_n$ and $s_1, \ldots, s_m$. The labelled matching DAG for Example 1 is shown in Figure 6.

5.2 The general case

We now show how the idea of labelled matching DAGs can be modified to handle terms with arbitrary nesting of associative function symbols and how such matching DAGs can be efficiently constructed.

Consider the general case of finding the set of matching block substitutions for linear $p_0 \in \widehat{T_\Sigma(X)}$ and $s_0 \in \overline{T_\Sigma(X)}$. We will actually generate an encoding $E(p_0, s_0)$ of this set. We will define this encoding inductively by considering some subterm p occurring in p_0 and some subterm s occurring in s_0. The encoding of the set of matching block substitutions for p and s will be denoted $E(p, s)$. This encoding will either consist of the special object *fail* indicating that no such matching block substitution exists or a pair $\langle S, G \rangle$ where S is a set of triples and G is a matching DAG.

Each triple in S will have one of three forms:

[5] Recall that in this simplified case B_k may not contain any associative function symbols

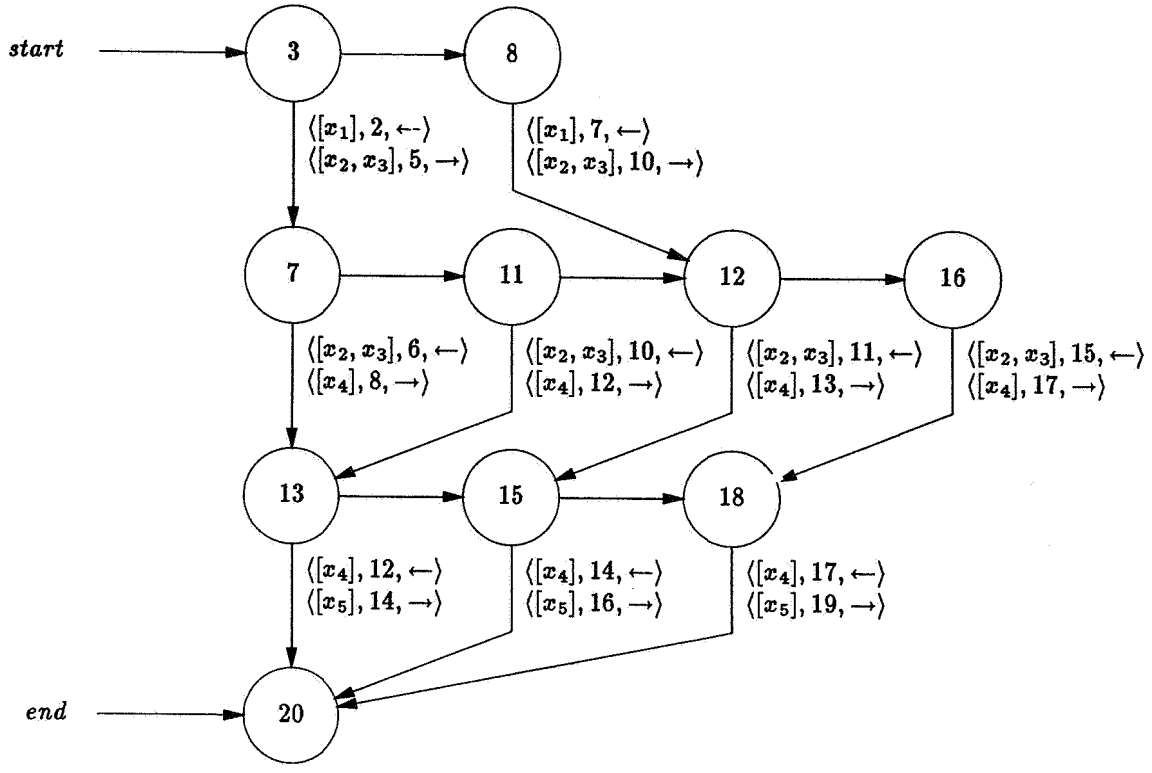

Figure 6: Labelled matching DAG for Example 1

1. $\langle B, \Gamma, \rightarrow \rangle$ where $B \in VB(p)$ and Γ is a position in s_0 that lies within s.
2. $\langle x, \Gamma, \leftrightarrow \rangle$ where $x \in NAV(p)$ and Γ is a position in s_0 that lies within s.
3. $\langle B, \Gamma, \leftarrow \rangle$ where $B \in VB(p)$ and Γ is a position in s_0 that lies within s.

The intuition is that these three forms of triple correspond in the obvious way to the three forms defined in §5.1. The difference here is that instead of having an index or a subterm as the second item we store a position relative to s_0. This change of representation is made so that a triple can be interpreted with respect to the whole subject term s_0 rather than just in the context where it is created.

We will use the symbol $\emptyset$ to denote the empty DAG with no nodes or edges. As in §5.1, each node in a DAG will have potentially two distinguished edges, called *across* and *down*. Each node will be labelled with an integers and each *down* edge may be labelled with a set of triples. The *end* node in a DAG never has outgoing edges.

We define the union $G_1 \cup G_2$ of two DAGs to be G_2 if $G_1 = \emptyset$, G_1 if $G_2 = \emptyset$ or a new DAG G otherwise. In this latter case the node set of G is the union of the nodes sets of G_1 and G_2 while the edge set of G is the union of the edge sets of G_1 and G_2 together with an extra unlabelled *down* edge from the *end* node of G_1 to the *start* node of G_2. The *start* node of G is the *start* node G_1 and the *end* node of G is the *end* node of G_2. Note that this definition of DAG union is associative but not commutative.

Definition 6 *The union of two encodings E_1, E_2 is given by*

$$E_1 \cup E_2 = \begin{cases} \mathit{fail} & \mathit{if}\ E_1 = \mathit{fail}\ \mathit{or}\ E_2 = \mathit{fail} \\ \langle S_1 \cup S_2, G_1 \cup G_2 \rangle & \mathit{if}\ E_1 = \langle S_1, G_1 \rangle\ \mathit{and}\ E_2 = \langle S_2, G_2 \rangle \end{cases}$$

Note that the union operation on encodings is associative but not commutative. We will use the notation $\bigcup_{i=1}^{n} E_i$ to mean $E_1 \cup E_2 \cup \cdots \cup E_n$.

We define $E(p,s)$ by induction on the structure of p. If $p \in X$ then $E(p,s) = \langle\{\langle p, \Gamma, \leftrightarrow\rangle\}, \emptyset\rangle$ where $s = s_0|_\Gamma$. If $p \in \Sigma_0$ then

$$E(p,s) = \begin{cases} \langle\emptyset,\emptyset\rangle & \text{if } p = s \\ \mathit{fail} & \text{if } p \neq s \end{cases}$$

If $p = f(p_1,\ldots,p_n)$ where $f \in \Sigma_n$, $f \notin \Sigma_A$ then unless $s = f(s_1,\ldots,s_n)$, $E(p,s) = \mathit{fail}$. Otherwise

$$E(p,s) = \bigcup_{i=1}^{n} E(p_i,s_i)$$

If $p = f^*(B_1,\ldots,B_q)$ where $f \in \Sigma$ then unless $s = f^*(s_1,\ldots,s_m)$ with $m \geq \sum_{i=1}^{q} l(B_i)$, $E(p,s) = \mathit{fail}$. Otherwise we consider a number of cases.

1. $q = 1$ and B_1 is a nonvariable block. Then

$$E(p,s) = \begin{cases} \bigcup_{i=1}^{m} E(p_i,s_i) & \text{if } B_1 = [p_1,\ldots,p_m] \\ \mathit{fail} & \text{if } l(B_1) \neq m \end{cases}$$

2. $q = 1$ and B_1 is a variable block. Then

$$E(p,s) = \langle\{\langle B_1, \Gamma.1, \rightarrow\rangle, \langle B_1, \Gamma.m, \leftarrow\rangle\},\rangle$$

   where $s = s_0|_\Gamma$.

3. $q > 1$ and both B_1 and B_q are variable blocks. Then

$$E(p,s) = E([B_1,\ldots,B_q],[s_1,\ldots,s_m])$$

   where the encoding $E([B_1,\ldots,B_q],[s_1,\ldots,s_m])$ of the set of matching block substitutions for a sequence of blocks $B_1,\ldots,B_q$ and a sequence of substitutions $s_1,\ldots,s_m$ is defined below.

4. $q > 1$ and both $B_1 = [p_1,\ldots,p_{l(B_1)}]$ and $B_q = [t_1,\ldots,t_{l(B_q)}]$ are nonvariable blocks. Then

$$E(p,s) = \left(\bigcup_{i=1}^{l(B_1)} E(p_i,s_i)\right) \cup \left(\bigcup_{i=1}^{l(B_q)} E(t_i, s_{m-l(B_q)+i})\right) \cup E([B_2,\ldots,B_{q-1}],[s_{1+l(B_1)},\ldots,s_{m-l(B_q)}])$$

5. $q > 1$, $B_1 = [p_1,\ldots,p_{l(B_1)}]$ is a nonvariable block and B_q is a variable block. Then

$$E(p,s) = \left(\bigcup_{i=1}^{l(B_1)} E(p_i,s_i)\right) \cup E([B_2,\ldots,B_q],[s_{1+l(B_1)},\ldots,s_m])$$

6. $q > 1$, B_1 is a variable block and $B_q = [t_1,\ldots,t_{l(B_q)}]$ is a nonvariable block. Then

$$E(p,s) = \left(\bigcup_{i=1}^{l(B_q)} E(t_i, s_{m-l(B_q)+i})\right) \cup E([B_1,\ldots,B_{q-1}],[s_1,\ldots,s_{m-l(B_q)}])$$

Finally $E([B_1,\ldots,B_n],[s_1,\ldots,s_m])$ where B_1 and B_n are variable blocks and $m \geq \sum_{i=1}^{n} l(B_i)$ is defined as follows. If $n = 1$ then

$$E([B_1],[s_1,\ldots,s_m]) = \langle\{\langle B_1, \Gamma, \rightarrow\rangle, \langle B_1, \Gamma', \leftarrow\rangle\}, \emptyset\rangle$$

where $s_1 = s_0|_\Gamma$ and $s_m = s_0|_{\Gamma'}$. Otherwise $n \geq 3$ and we define a pair $\langle S, G\rangle$ where G is built along similar lines to the construction of a labelled DAG in §5.1. However we have one major difference in that the blocks $B_1,\ldots,B_n$ may themselves contain associative function symbols.

For $k \in \{2, 4, \ldots, n-3, n-1\}$ we define

$$Ind_k = \left\{ \alpha \mid B_k = [p_1, \ldots, p_{l(B_k)}] \text{ and } \bigcup_{i=1}^{l(B_k)} E(p_i, s_{\alpha+i-1}) \neq \mathit{fail} \text{ and } \alpha > \sum_{i=1}^{k-1} l(B_i) \text{ and } \alpha + \sum_{i=k}^{n} l(B_i) \leq m+1 \right\}$$

For each set Ind_k we have a set $Node_k$ of nodes with a node labelled α in $Node_k$ for each $\alpha \in Ind_k$. We also have a singleton set $Node_{n+1}$ containing a node labelled $m+1$.

Now for $k \in \{2, 4, \ldots, n-3, n-1\}$, each node $v \in Node_k$ other than the one with the highest label has an *across* edge going to the node in $Node_k$ with the next highest label. Each node $v \in Node_k$ also has a *down* edge iff there exists a node in $v' \in Node_{k+2}$ with least label such that $label(v') \geq label(v) + l(B_k) + l(B_{k+1})$. However the target and label of this down edge depend on the encoding for the block substitutions that match B_k to $s_\alpha, \ldots, s_{\alpha+i-1}$. Suppose $B_k = [p_1, \ldots, p_{l(B_k)}]$ and let

$$\langle S_k, G_k \rangle = \bigcup_{i=1}^{l(B_k)} E(p_i, s_{\alpha+i-1}).$$

The *down* edge from v has the label

$$S_k \cup \{\langle B_{k-1}, \Gamma, \leftarrow\rangle, \langle B_{k+1}, \Gamma', \rightarrow\rangle\}$$

where $s|\Gamma = s_{\alpha-1}$ and $s|\Gamma' = s_{\alpha+l(B_k)}$. If $G_k = \emptyset$ then this edge goes to v'. Otherwise if $G_k \neq \emptyset$ then this edge goes to the *start* node of G_k and we have an extra (unlabelled) *down* edge from the *end* node of G_k to v'.

Let G' be the DAG thus formed, with the *start* node being the node in N_2 with least label and the *end* node being the single node in N_{n+1}. We form a new DAG G by removing all nodes from G' that do not lie on a path from the *start* node to the *end* node. If G is the empty DAG then

$$E([B_1, \ldots, B_n], [s_1, \ldots, s_m]) = \mathit{fail}$$

Otherwise

$$S = \{\langle B_1, \Gamma, \rightarrow\rangle, \langle B_n, \Gamma', \leftarrow\rangle\}$$

where $s|\Gamma = s_1$ and $s|\Gamma' = s_m$. Then

$$E([B_1, \ldots, B_n], [s_1, \ldots, s_m]) = \langle S, G \rangle$$

Once we have $E(p_0, s_0)$ we can extract individual matching block substitutions as follows. If $E(p_0, s_0) = \mathit{fail}$ then there is not a block substitution ϕ such that $p_0\phi = s_0$. Otherwise if $E(p_0, s_0) = \langle S, G \rangle$ then for each path from the *start* node in G to the *end* node in G, we form the union S' of all the sets of triples occurring as labels to edges in the path, together with S. (If $G = \emptyset$ then we have a single block substitution corresponding to $S' = S$.) The corresponding block substitution for ϕ for each such S' is defined by

$$\phi_V(x) = s_0|_\Gamma \quad \text{where } \langle x\Gamma, \leftrightarrow\rangle \in S' \text{ and } x \in NAV(p_0).$$

$$\phi_B(B) = s_0|_{\Gamma.\alpha}, \ldots, s_0|_{\Gamma.\beta} \quad \text{where } \langle B, \Gamma.\alpha, \rightarrow\rangle, \langle B, \Gamma.\beta, \leftarrow\rangle \in S' \text{ and } B \in VB(p_0).$$

An efficient algorithm for constructing this encoding is given as pseudo code in Figures 7,8 and 9. Here we represent a DAG G, by it's *start* and *end* nodes[6] $G.start$ and $G.end$. Since only *down* edges have labels and each node has at most one down edge, the label of each edge can be stored in the source node. Accordingly each node is represented by

[6] We can reach all other nodes by tracing edges from the *start* node but it is convenient to be able to access the *end* node in constant time.

SOLVE(x, s, Γ) **where** $x \in X$ **is**
    return($\langle\{\langle x, \Gamma, \leftrightarrow\rangle\}, \emptyset\rangle$).

SOLVE(c, s, Γ) **where** $c \in \Sigma_0$ **is**
    if $s = c$ **then**
        return($\langle\emptyset, \emptyset\rangle$)
    else
        return(*fail*)
    fi.

SOLVE($f(p_1, \ldots, p_n), s, \Gamma$) **where** $f \in \Sigma_n, f \notin \Sigma_A$ **is**
    if $s = f(s_1, \ldots, s_n)$ **then**
        return(SOLVE_SEQ($[p_1, \ldots, p_n], [s_1, \ldots, s_n], \Gamma.1$))
    else
        return(*fail*)
    fi.

SOLVE($f^*(B_1, \ldots, B_q), s, \Gamma$) **where** $f \in \Sigma_A$ **is**
    if $s = f^*(s_1, \ldots, s_m)$ **and** $m \geq \sum_{i=1}^{q} l(B_i)$ **then**
        if NONVAR(B_1) **then**
            if $q = 1$ **then return**(SOLVE_SEQ($B_1, [s_1, \ldots, s_m], \Gamma.1$)) **fi**;
            left_sol := SOLVE_SEQ($B_1, [s_1, \ldots, s_{l(B_1)}], \Gamma.1$);
            if *left_sol* = *fail* **then return**(*fail*) **fi**;
            first_var := 2; *first_sub* := $l(B_1) + 1$
        else
            left_sol := $\langle\emptyset, \emptyset\rangle$; *first_var* := 1; *first_sub* := 1
        fi;
        if NONVAR(B_q) **then**
            right_sol := SOLVE_SEQ($B_q, [s_{m-l(B_q)+1}, \ldots, s_m], \Gamma.(m - l(B_q) + 1)$);
            if *right_sol* = *fail* **then return**(*fail*) **fi**;
            last_var := $q - 1$; *last_sub* := $m - l(B_q)$
        else
            right_sol := $\langle\emptyset, \emptyset\rangle$; *last_var* := q; *last_sub* := m
        fi;
        if *first_var* = *last_var* **then**
            return(FUSE(FUSE(*left_sol*, *right_sol*), $\{\langle B_{first\_var}, \Gamma.(first\_sub), \rightarrow\rangle$,
                $\langle B_{first\_var}, \Gamma.(last\_sub), \leftarrow\rangle\}$))
        fi;
        t = BUILD_DAG($[B_{first\_var}, \ldots, B_{last\_var}], [s_{first\_sub}, \ldots, s_{last\_sub}], \Gamma.(first\_sub)$);
        if t = *fail* **then**
            return(*fail*)
        else
            return(FUSE(FUSE(*left_sol*, *right_sol*), t)))
        fi
    else
        return(*fail*)
    fi.

Figure 7: Algorithm 3 (main routine).

```
BUILD_DAG([B_1, ..., B_n], [s_1, ..., s_m], Γ.k) is
    α = 1; β := 1 + m − Σ_{i=1}^{n} l(B_i); N_2 := NEW_NODE();
    for i := 2 to n − 1 step 2 do (* Phase 1 *)
        α := α + l(B_{i−1}) − 1; β := β + l(B_{i−1});
        repeat
            α := α + 1;
            if α > β then return(fail) fi;
            t := SOLVE_SEQ(B_i, [s_α, ..., s_{α+l(B_i)−1}], Γ.(k + α − 1))
        until t ≠ fail;
        N_{i+2} = NEW_NODE();
        INSERT_SOL(t, N_i, N_{i+2}, α, {⟨B_{i−1}, Γ.(k + α − 2), ←⟩, ⟨B_{i+1}, Γ.(k + α + l(B_i) − 1), →⟩});
        α := α + l(B_i); β := β + l(B_i)
    od;
    N_{n+1}.down_label := ∅; N_{n+1}.label := m + 1; N_{n+1}.across := nil; N_{n+1}.down := nil;

    for i := n − 1 downto 2 step 2 do (* Phase 2 *)
        V := N_i; L := N_{i+2}; α := N_i.label; finished := false;
        repeat
            α := α + 1;
            while α + l(B_i) + l(B_{i+1}) > L.label and finished = false do
                if L.across = nil then finished := true else L := L.across fi
            od
            if finished = false then
                t := SOLVE_SEQ(B_i, [s_α, ..., s_{α+l(B_i)−1}], Γ.α);
                if t ≠ fail then
                    V.across := NEW_NODE(); V := V.across;
                    INSERT_SOL(t, V, L, α, {⟨B_{i−1}, Γ.(k + α − 2), ←⟩, ⟨B_{i+1}, Γ.(k + α + l(B_i) − 1), →⟩})
                fi
            fi
        until finished = true;
        V.across := nil
    od;
    S := {⟨B_1, Γ.k, →⟩, ⟨B_n, Γ.(k + m − 1), ←⟩} ;
    G.start = N_2; G.end = N_{n+1};
    return(⟨S, G⟩).
```

Figure 8: Algorithm 3 (subroutine to build DAG).

$v.label$	Integer label.
$v.across$	Target of v's *across* edge or **nil** if no such edge exists.
$v.down$	Target of v's *down* edge or **nil** if no such edge exists.
$v.down\_label$	Set of triples labelling v's *down* edge or **nil** if the edge or label does not exist.

The overall structure of Algorithm 3 resembles that of Algorithm 1, however the details are naturally governed by the definition of $E(p_0, s_0)$. Note that we need an extra parameter Γ to track the current position in the original subject term s_0.

The subroutine SOLVE_SEQ computes the union of the encodings for a sequence of pattern and subject terms. The subroutine FUSE computes the union of two encodings where it is known that neither is *fail*. The subroutine BUILD_DAG (which corresponds to the subroutine SLIDE in Algorithm 1) is where most of the real work takes place. The subroutine INSERT_SOL is used when building a DAG to insert a encoding computed for a subproblem in between the nodes N and N' by either inserting just a *down* edge or a whole subgraph.

```
SOLVE_SEQ([p_1, ..., p_n], [s_1, ..., s_m], Γ.k) is
    if n ≠ m then return(fail) fi;
    r := ⟨∅, ∅⟩;
    for i := 1 to n do
        t := SOLVE(p_i, q_i, Γ.(k + i − 1));
        if t = fail then return(fail) fi;
        r := FUSE(r, t)
    od;
    return(r).

FUSE(⟨S_1, G_1⟩, ⟨S_2, G_2⟩) is
    if G_1 = ∅ then
        G := G_2
    else
        if G_2 = ∅ then
            G := G_1
        else
            G.start := G_1.start; G.end := G_2.end; G_1.end.down := G_2.start
        fi
    fi;
    return(⟨S_1 ∪ S_2, G⟩).

INSERT_SOL(⟨S, G⟩, N, N', α, T) is
    if G ≠ ∅ then
        N.down := G.start; N'.end.down := L
    else
        N.down := N'
    fi;
    N.label := α; N.down_label := S ∪ T;
    return
```

Figure 9: Algorithm 3 (other subroutines).

For efficiencies sake, the DAG is constructed in reduced form from the outset; i.e. only nodes which lie on a path from *start* to *end* are ever generated. We assume the existence of a subroutine NEW_NODE which returns fresh dynamically allocated nodes. The subroutine BUILD_DAG consists of two phases.

The first phase works from B_2 to B_{n-1} and constructs all the nodes on the single[7] path from the (eventual) *start* node of G to the (eventual) *end* node of G. In this phase each block B_k for $k \in \{2, 4, \ldots, n-3, n-1\}$ is matched at the earliest position in $s_1, \ldots, s_n$. This forms the leftmost path in the DAG (if a path exists at all) and is effectively the matches found (implicitly) by Algorithm 1. This path contains no *across* edges.

The second phase works from B_{n-1} to B_2 and for each block B_k for $k \in \{2, 4, \ldots, n-3, n-1\}$ it generates the remaining nodes in $Node_k$ that occur on a *start* to *end* path. This condition is easily checked since for $v \in Node_k$ we only have to find a node $v' \in Node_{k+2}$ that has already be created and such that $v'.label \geq v.label + l(B_k) + l(B_{k+1})$. Such a node can be found (if it exists) by one pass through the nodes of N_{k+2}. But this pass is actually be done incrementally just once for *all* the nodes in N_k.

Example 2 *Consider the pattern*

$$p = f^*([h(x_1)], [x_2], [g^*([h(x_3)], [x_4])], [x_5], [g^*([x_6], [a], [x_7])], [x_8])$$

[7] Actually there may be a number of paths created in the first phase due to the inclusion of subgraphs resulting from matching pattern subterms from a nonvariable block against subject terms.

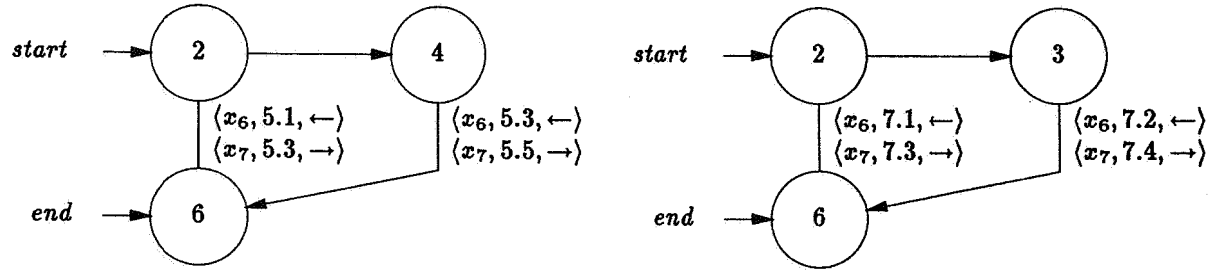

Figure 10: Matching sub-DAGs formed for Example 2.

and the subject

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

Applying Algorithm 3 we form two sub-DAG's which correspond to matching the nonvariable block $[g^*([x_6], [a], [x_7])]$ *against subject terms* $g^*(h(a), a, b, a, b)$ *and* $g^*(b, a, a, b, c)$. *These are shown in Figure 10. The complete encoding for this example is the set*

$$S = \{\langle x_1, 1.1, \leftrightarrow\rangle, \langle x_2, 2, \rightarrow\rangle, \langle x_8, 8, \leftarrow\rangle\}$$

and the DAG G shown in Figure 11.

5.3 Implementation and Complexity

For efficient implementation, positions (i.e. strings of integers) and occurrences of variable blocks within triples are replaced by pointers. Thus the storage requirements of a triple are constant. Sets of triples may be represented simply as linked lists where pointers to both the head and tail are kept. This allows set union to be computed in constant time.

Theorem 8 *Algorithm 3 running on pattern p and subject s takes* $O(|p|.|s|)$ *time.*

Proof: Let the time taken by Algorithm 3 running on pattern p and subject s be $cost(p, s)$. We first obtain an expression to bound this cost in each of the four cases.

Case 1: $p \in X$
Clearly $cost(p, s) = K_1$ for some constant K_1.

Case 2: $p \in \Sigma_0$
Clearly $cost(p, s) = K_2$ for some constant K_2.

Case 3: $p = f(p_1, \ldots, p_n)$ for $f \in \Sigma_n$, $f \notin \Sigma A$
Then $cost(p, s)$ is maximised when $s = f(s_1, \ldots, s_n)$ and the algorithm could call itself (via SOLVE_SEQ) on each pair (p_i, s_i) for $i \in \{1, \ldots, n\}$. We have

$$cost(p, s) \leq K_3 n + \sum_{i=1}^{n} cost(p_i, s_i)$$

for some constant K_3.

Case 4: $p = f^*(B_1, \ldots, B_q)$ for $f \in \Sigma_A$
Then $cost(p, s)$ is maximised when $s = f^*(s_1, \ldots, s_m)$ and $m \geq \sum_{i=1}^{q} l(B_i)$ and the algorithm may have to construct a DAG. Suppose in the maximally flattened form of p the argument list of f^* is $p_1, \ldots, p_n$. First note that the algorithms calls itself (via BUILD_DAG and SOLVE_SEQ at most once on each pair of terms p_i, s_j. In phase 1 of BUILD_DAG we have two loop nested loops where the outer one iterates over

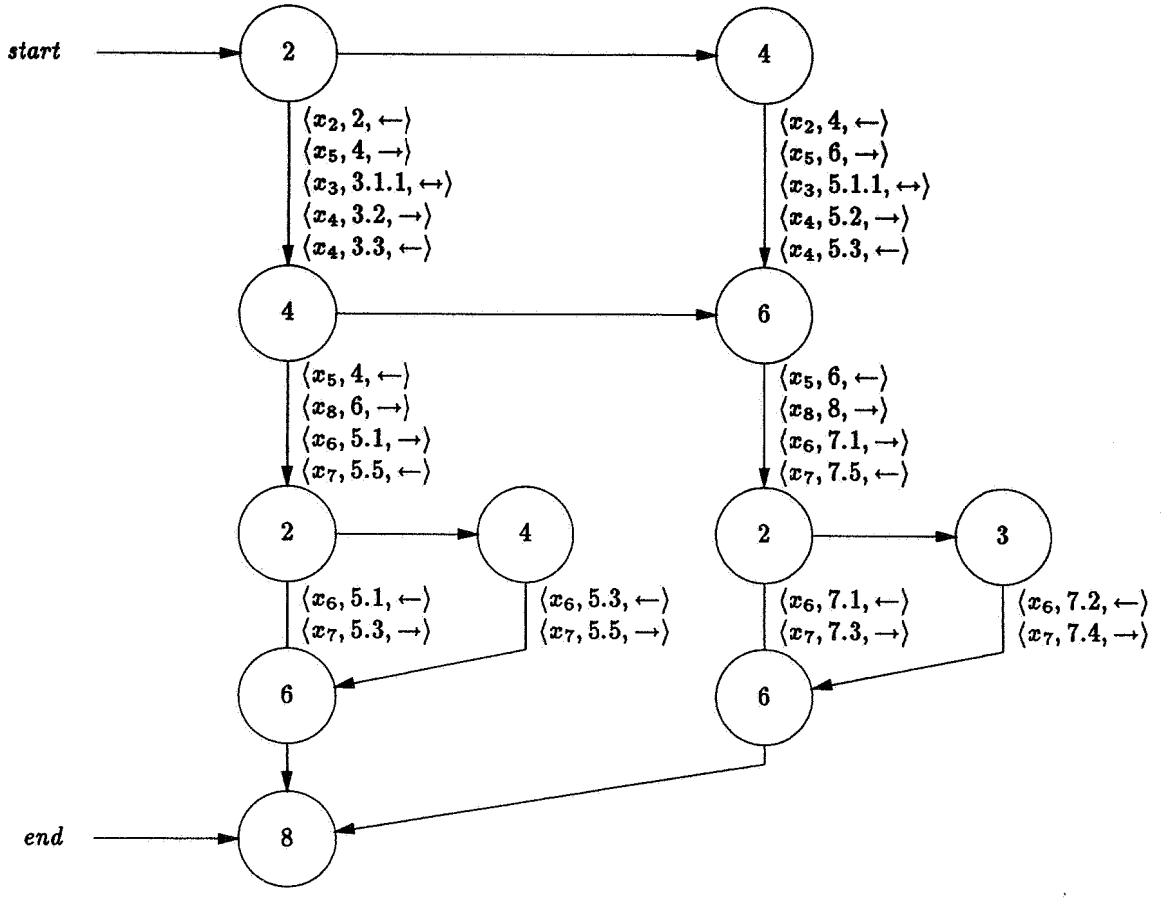

Figure 11: Matching DAG for Example 2.

blocks and the inner one iterates over subject terms. Calls to SOLVE_SEQ from inside these loops cause iteration over the length of each nonvariable block. Thus the computation required excluding recursive calls is proportional to nm. A similar argument holds for phase 2 since as noted previously the innermost while loop only iterates through each set of nodes of N_k at most once and each $|N_k|$ is bounded by m. Thus we have

$$cost(p, s) \leq K_4 nm + \sum_{i=1}^{n} \sum_{j=1}^{m} cost(p_i, s_j)$$

for some constant K_4. The remainder of the proof is identical to that of Theorem 3. □

Theorem 9 *Each matching block substitution for p and s can be extracted from $E(p, s)$ in $O(Var(p))$ time if they are extracted in a particular order and pointers are used to avoid copying subterms.*

Proof: We can divide the time require to extract a matching block substitution from $E(p, s)$ into two parts—the time to traverse a path through the DAG and time to deal with the variable bindings.

Taking the latter first an easy way to deal with variable bindings is to associate a piece of storage with each variable in $NAV(p)$ and each variable block in $VB(p)$ and to copy the subject pointer (i.e. the second item) of each triple into each the appropriate piece of storage as the triple is encountered. Clearly this takes constant time and the number of triples for any one block substitution is bounded by $2|Var(p)|$.

Getting the same complexity bound on traversing a path through the DAG is more complicated since any particular path could visit a number of nodes proportional to $|s|$. However the number of *down* edges on any path is bounded by $|Var(p)| - 1$. Notice that from every node in the DAG we can always reach the *end* node by taking only *down* edges. Now if we extracted all paths from the *start* node to the *end* node using a straight forward recursive search that follows *down* edges before following *across* edges each path successive path contains exactly one *across* edge that did not occur in the previous path. Thus the number of edges traversed in following the new path from the point where it diverged from the old path is bounded by $|Var(p)|$. In this way the cost of traversing long sequences of *across* edges is effectively shared out amongst successive paths. □

6 Discussion

Recall that when in Algorithms 1 and 3 we search for a subsequence w from $s_1, \ldots, s_m$ such that a nonvariable block $B = p_1, \ldots, p_n$ matches w we may in the worst case have to try matching almost every pattern term p_i against almost every subject term s_j. Now Algorithm 3 has intrinsically quadratic worst case time complexity because we build a data structure which could have quadratic size. However in Algorithm 1 it is the need to try matching almost every pattern pattern terms p_i against almost every subject term s_j that accounts for the quadratic worst case running time. It is therefore natural to ask whether the techniques that reduce the quadratic worst case running time in the naive string matching algorithm to the linear time required by algorithms such as Knuth-Morris-Pratt [8] can be applied in this situation. We now explain why this appears to be difficult.

Let R be a set of objects with a relation $\succeq$ to model the notion of matching between objects. Suppose we have a pattern $P = p_1, \ldots, p_n \in R^+$ and a subject $S = s_1, \ldots, s_m \in R^+$ and we wish to find a subsequence $S' = s_\alpha, \ldots, s_{\alpha+n-1}$ such that P matches S, i.e. for $i \in \{1, \ldots, n\}$, $p_i \succeq s_{\alpha+i-1}$. Now fast string matching algorithms rely on the following property: For all $p, p' \in R$ either

$$\forall s \in R.[p \succeq s \Rightarrow p' \succeq s] \tag{1}$$

or

$$\forall s \in R.[p \succeq s \Rightarrow p' \not\succeq s] \tag{2}$$

holds. The importance of this property is that for each object s in the subject once we have found some p in that pattern such that p matches s we know for every other object p' in the pattern whether p' matches s. Clearly in the string matching case where R is an alphabet and $\succeq$ is equality the property holds trivially. However in Algorithm 1, we have $R = \overline{T_\Sigma(X)}$ and $\succeq$ is associative matching.

Definition 7 *Two terms t, t' are said to be semi-unifiable modulo E if there exists a pair of substitutions $\sigma, \sigma' : X \to T_\Sigma(X)$ such that $t\sigma =_E t'\sigma'$.*

Now (1) above corresponds to p' matching p modulo associativity and (2) above corresponds to p not being semi-unifiable with p' modulo associativity. Now there are pairs of terms p, p' (which need not even involve associative function symbols) such that neither (1) nor (2) holds.

Example 3 *Let $p = f(x_1, a)$ and $p' = f(a, x_2)$ where $f \notin \Sigma_A$. Now clearly p and p' are semi-unifiable modulo associativity but p' does not match p modulo associativity.*

Thus we face two problems; firstly in order to preprocess the pattern as required by Knuth-Morris-Pratt we would need to decide associative semi-unification (on linear terms) and secondly even then we might not get a linear time algorithm for the remainder of the task because we might encounter a pair p, p' in a nonvariable block for which neither (1) nor (2) held.

We now briefly consider the associative matching problem for nonlinear pattern terms. Every nonlinear term t can be linearized to a term t' by replacing repeated occurrences of a variable with a fresh unused variable. In order that information is not lost we construct a set I_t which contains a pair $\langle x_i, x_j \rangle$ for each occurrence of x_i that is replaced by x_j. For example if $t = f(g(f(x_1, a), f(x_2, x_2)), f(g(x_3, x_1), b))$ we could linearize t to $t' = f(g(f(x_1, a), f(x_2, x_4)), f(g(x_3, x_5), b))$ where $I_t = \{\langle x_2, x_4 \rangle, \langle x_1, x_5 \rangle\}$. Now give a subject s, for each substitution σ such that $t'\sigma =_A s$ we have

$$t\sigma =_A s \Leftrightarrow \text{for all } \langle x_i, x_j \rangle \in I_t,\ \sigma(x_i) =_A \sigma(x_j) \tag{3}$$

Thus one way of solving the associative matching problem for $p, s \in T_\Sigma(X)$ where p is a nonlinear term is to linearize p, apply Algorithm 3 to obtain an encoding of all the matching block substitutions and then check each flattened substitution against the right hand side of (3) after it is extracted. This simple scheme can be improved somewhat by combining the checking of flattened substitutions with their extraction.

Recall that extracting a flattened substitution from an encoding is a two step process. First we extracted a block substitution corresponding to a path in the DAG part of the encoding and then we construct the induced flattened substitutions by considering the various ways of splitting the sequence of subject terms assigned to each variable block among the variables in that variable block. In the first step as we choose a path through the DAG we can perform a check every time we select an *down* edge whose label includes a triple $\langle x_i, \Gamma, \leftrightarrow \rangle$ to see if there exists a pair $\langle x_i, x_j \rangle$ or $\langle x_j, x_i \rangle$ in I_p such that x_j has been assigned some (flattened) term not equal to $s|_\Gamma$ as a result of choosing a previous *down* edge. By using appropriate pointers, this test requires at worst a comparison of two flattened subterms for equality. If we find a conflict we can abandon *all* paths that start with our current choice of edges without explicitly generating them. In the second step we can do a similar check for conflicts each time we split a sequence of subject subterms among the variables in a variable block. Again if we find a conflict we can abandon all induced substitutions that include our current choice of splits without explicitly generating them. Thus we cut down the search space of potential flattened substitutions which might satisfy the right hand side of (3).

Acknowledgement

I thank Arie van Deursen, Jan Heering, Jasper Kamperman and Paul Klint for their comments on drafts of this paper.

References

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

[2] R. M. Gallimore, D. Coleman, and V. Stavridou. UMIST OBJ: a language for executable program specifications. *Computer Journal*, 32(5):413–421, 1989.

[3] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, August 1988.

[4] P. R. H. Hendriks. Lists and associative functions in algebraic specifications. Technical Report CS-R8908, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[5] P. R. H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[6] J. M. Hullot. Associative commutative pattern matching. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 406–412, 1979.

[7] D. Kapur and P. Narendran. Np-completeness of the set unification and matching problems. In *Proceedings of the 8th International Conference on Automated Deduction*, Lecture Notes in Computer Science 230, pages 489–495. Springer-Verlag, 1986.

[8] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[9] E. Kounalis and D. Lugiez. Compilation of pattern matching with associative-commutative func tions. In A. Abramsky and T. S. E. Maibaum, editors, *Proceedings of TAPSOFT '91*, Lecture Notes in Computer Science 493, pages 57–73. Springer-Verlag, 1991.

[10] Patrick Lincoln and Jim Christian. Adventures in associative-commutative unification. *Journal of Symbolic Computation*, 8:217–240, 1989.

[11] E. W. Myers. An applicative random access stack. *Information Processing Letters*, 17:241–248, 1983.

[12] K. U. Schulz, editor. *Word Equations and Related Topics.* Lecture Notes in Computer Science 572. Springer-Verlag, 1992.

[13] Mark E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the ACM*, 28(3):423–434, 1981.