Optimal Left-to-Right Pattern-Matching Automata

Nadia Nedjah[*], Colin D. Walter and Stephen E. Eldridge

Computation Dept., UMIST, PO. Box 88, Manchester M60 1QD, UK.
nn, cdw, see @sna.co.umist.ac.uk www.co.umist.ac.uk

Abstract

We propose a practical technique to compile pattern-matching for prioritised overlapping patterns in equational languages into a minimal, deterministic, left-to-right, matching automaton. First, we present a method for constructing a tree matching automaton for such patterns. This allows pattern-matching to be performed without any backtracking. Space requirements are reduced by using a directed acyclic graph (*dag*) automaton that shares all the isomorphic subautomata which are duplicated in the tree automaton. We design an efficient method to identify such subautomata and avoid duplicating their construction while generating the dag automaton. We conclude with some easily computed bounds on the size of the automata, thereby improving on previously known equivalent bounds for the tree automaton.

Keywords: Term rewriting system, pattern-matching, tree automaton, dag automaton.

1. Introduction

The key technical problems in implementing equational computations as reduction sequences are finding redexes, choosing which redex to reduce at each step and performing the reduction. The first stage is often called *pattern-matching* and this can be achieved as in lexical analysis by using a finite automaton. In order to avoid backtracking over symbols already read, extra patterns are added. These correspond to overlaps in the prefixes of original patterns. In this paper, we focus on avoiding the duplication of subautomata caused by the repetition of pattern suffixes when these new patterns are added. This results in a more efficient pattern-matcher. Consequently we also obtain much improved bounds on the size of the recognising automaton.

A simple-minded way of pattern-matching is to try each rule sequentially until a left-hand side is matched or the whole pattern set is exhausted. However, this method may consume considerable effort unnecessarily. Usually, patterns are pre-processed to produce an intermediate representation allowing the matching to be performed more efficiently. One such representation consists of a matching automaton [3, 4, 7].

Here we concentrate on another method of building minimal (with respect to size), deterministic (i.e. no backtracking), matching automata for prioritised, overlapping patterns. First, a method for generating a deterministic tree matching automaton for a given pattern set is described. Although the generated automaton is efficient since it avoids symbol re-examination, it can only achieve this at the cost of increased space. As we shall see, the main reason for the increase in space requirements is the duplication of functionally identical or isomorphic subautomata in the tree-based automaton. However, the problem of directly constructing matching automata that do not duplicate such subautomata has not been previously described [2, 11]. We tackle

[*] Author sponsored by the British Council and the Algerian Ministry for High Education.

this next, describing a method that efficiently identifies equivalent states that would lead to identical subautomata and then constructing the equivalent reduced dag-based automaton. This is achieved without explicitly constructing the tree automaton first. Finally, we bound the size of the dag automaton using easily obtained parameters.

2. Notation and Definitions

In this section, we recall the notation and concepts that will be used in the rest of the paper. Symbols in a *term* are either function or variable symbols. The non-empty set of function symbols $F = \{a, b, f, g, ...\}$ is *ranked* i.e., every function symbol f in F has an *arity* which is the number of its arguments and is denoted $\#f$. A term is either a constant, a variable or has the form $ft_1\, t_2...t_{\#f}$ where each t_i, $1 \le i \le \#f$, is itself a term. We abbreviate terms by removing the usual parenthesis and commas. This is unambiguous in our examples since the function arities will be kept unchanged throughout, namely $\#f = 4$, $\#g = \#h = 2$, $\#a = \#b = 0$. Variable occurrences are replaced by ω, a meta-symbol which is used since the actual symbols are irrelevant here. A term containing no variables is said to be a *ground* term. We generally assume that patterns are linear terms, i.e. each variable symbol can occur at most once in them. Pattern sets will be denoted by L and patterns by $\pi 1$, $\pi 2$, ..., or simply by π. A term t is said to be an *instance* of a pattern π if there exists a *substitution* σ for the variables of π such that $t = \sigma\pi$.

Definition 2.1: A *position* in a term is a path specification which identifies a node in the parse tree of the term. Position is specified here using a list of positive integers. The empty list Λ denotes the position of the root of the parse tree and the position $p.k$ ($k \ge 1$) denotes the root of the kth argument of the function symbol at position p.

Positions of symbols in a term can be totally ordered according to the left-to-right order of the symbols in the term or the pre-order traversal of the parse tree. This ordering generalises to cases where the positions are of symbols in different terms because we can compare their integer lists lexicographically. So any set of positions can be put in left-to-right order.

Definition 2.2: A *matching item* is a triple $r{:}\alpha\bullet\beta$ where $\alpha\beta$ is a term and r is a *rule label*. The label identifies the origin of the term $\alpha\beta$ and hence, in a term rewriting system, the rewrite rule which has to be applied when $\alpha\beta$ is matched. The label is not written explicitly below except where necessary. The meta-symbol • is called the *matching dot*, α and β are called the *prefix* and *suffix* respectively, and the first symbol of β is called the *matching symbol*. The position of the matching dot is called the *matching position* and is identified with the position of the matching symbol. A *final* matching item is one of the form $\alpha\bullet$. It has a *final* matching position which we write as ∞.

Throughout this paper left-to-right traversal order is used. So the matching item $\bullet\beta$ represents the initial state prior to matching the pattern β. In general, the matching item $\alpha\bullet\beta$ denotes that the symbols in α have been matched and those in β have not yet been recognised. Finally, the matching item $\alpha\bullet$ is reached on successfully matching the whole pattern α.

Definition 2.3: A set of matching items in which all the items have the same prefix is called a *matching set*. A matching set in which all the items have an empty prefix is called an *initial* matching set whereas a matching set in which all the items have an empty suffix is called a *final* matching set. The *rule set* for a matching set is the set of labels appearing in its items.

Definition 2.4: For a set L of pattern suffixes and any symbol s, let $L\backslash s$ denote the set of pattern suffixes obtained by removing the initial symbol s from those members of L which commence with s. Then, for $f \in F$ define L_ω and L_f by:

$$L_\omega = L\backslash\omega$$

$$L_f = \begin{cases} L\backslash f \cup \omega^{\#f} L\backslash\omega & \text{if } L\backslash f = \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

where $\omega^{\#f}$ denotes a string of $\#f$ symbols ω. The *closure* $\overline{L}$ of a pattern set L is then defined recursively by Gräf [2] as follows:

$$\overline{L} = \begin{cases} L & \text{if } L = \{\varepsilon\} \text{ or } L = \varnothing \\ \bigcup_{s \in F \cup \{\omega\}} s\overline{L_s} & \text{otherwise} \end{cases}$$

Roughly speaking, with two item suffixes of the form $f\alpha$ and $\omega\beta$ we always add the suffix $f\omega^{\#f}\beta$ in order to postpone by one more symbol the decision between these two patterns. Otherwise backtracking might be required to match $\omega\beta$ if input f leads to failure to match $f\alpha$.

3. Tree Matching Automata

In this section, we describe a practical and efficient method to construct a tree matching automaton for a prioritised overlapping pattern set. The pattern set L is converted into the above closed pattern set $\overline{L}$ while generating the matching automaton. In general, the construction technique described here is inspired by the LALR method used in YACC to generate parsers for LR-languages [1,5]. This has been used for many years to compile imperative languages. The pattern set to be compiled is considered as a set of right-hand sides of syntactic productions. However, there are no Shift-Reduce or Reduce-Reduce conflicts since we are only treating root matching of the input and the priority rule enables us to resolve multiple matches.

The pattern set compiles into a deterministic tree matching automaton which is represented by the 4-tuple $\langle S_0, S, S_\infty, \delta\rangle$ where S is the state set, $S_0 \in S$ is the initial state, $S_\infty \subseteq S$ is the final state set and δ is the state transition function. The states are labelled by matching sets which consist of original patterns whose prefixes match the current input prefix, together with extra instances of the patterns which are added to avoid backtracking in reading the input. In particular, the matching set for S_0 contains the initial matching items formed from the original patterns and labelled by the rules associated with them. Transitions are considered according to the symbol at the matching position, i.e. that immediately after the matching dot. For each symbol $s \in F \cup \{\omega\}$ and state with matching set M, a new state with matching set $\delta(M,s)$ is derived using the composition of the functions *accept* and *close* defined in Figure 3.1.

$$\begin{array}{lll} accept(M,s) & = & \{ r{:}\alpha s\bullet\beta \mid r{:}\alpha\bullet s\beta \in M \} \\ close(M) & = & M \cup \{ r{:}\alpha\bullet f\omega^{\#f}\mu \mid r{:}\alpha\bullet\omega\mu \in M \text{ and} \\ & & \qquad \exists\, q{:}\alpha\bullet f\lambda \in M \text{ for some suffix } \lambda \text{ and } f \in F \} \\ \delta(M,s) & = & close(\ accept(M,s)\ ) \end{array}$$

Figure 3.1: *Automata Transition Function*

The items obtained by recognising the symbols in those patterns of M where s is the next symbol form the set $accept(M,s)$ which is called the *kernel* of $\delta(M,s)$. However, the set $\delta(M,s)$ may contain more items. The presence of two items $\alpha\bullet\omega\mu$ and $\alpha\bullet f\lambda$ in M creates a non-deterministic situation since the variable ω could be matched by a term having f as head symbol. The item $\alpha\bullet f\omega^{\#f}\mu$ is added to remove this non-determinism and avoid backtracking. The transition function thus implements simply the main step in the closure operation described by Gräf [2] and set out in the previous section. Hence the pattern set resulting from the automaton construction using the transition function of Figure 3.1 coincides with the closure operation of Definition 2.4. The item labels simply keep account of the originating pattern for when a successful match is achieved.

Non-determinism is worst where the input can end up matching the whole of two different patterns. Then we need a priority rule to determine which pattern to select.

Definition 3.2: A pattern set L is *overlapping* if there is a ground term that is an instance of at least two distinct patterns in L. Otherwise, L is *non-overlapping*.

Definition 3.3: A *priority rule* is a partial ordering on patterns such that if π_1 and π_2 are overlapping patterns then either π_1 has *higher* priority than π_2 or π_2 has *higher* priority than π_1.

When a final state is reached, if several rules have been successfully matched, then the priority rule is engaged to select the one of highest priority. Examples of priority rules are the *textual* and *specificity* priority rules. The textual rule is used in the majority of functional languages. Among the matched patterns, the rule chooses the pattern that appears first in the text. The specificity rule [6] can be used only if for any pair of overlapping patterns, one pattern is an instance of the other. The former pattern is said to be *more defined* than the latter. So among the matched patterns, the rule chooses the most defined pattern. Whatever rule is used, we will apply the word *match* only to the pattern of highest priority which is matched.

Definition 3.4: A term *t matches* a pattern $\pi \in L$ if, and only if, t is an instance of π and t is not an instance of any other pattern in L of higher priority than π.

Example 3.5: Let $L = \{faa\omega a,\ fga\omega aab,\ f\omega bbb\}$ be the set of patterns of rules numbered 1, 2 and 3 respectively. The matching automaton for L is given in Figure 3.6. Each state is labelled with its matching set. Transitions corresponding to failures are omitted, and an ω-transition is only taken when there is no other available transition which accepts the current symbol. The automaton can be used to drive the pattern-matching process irrespective of the chosen term rewriting strategy.

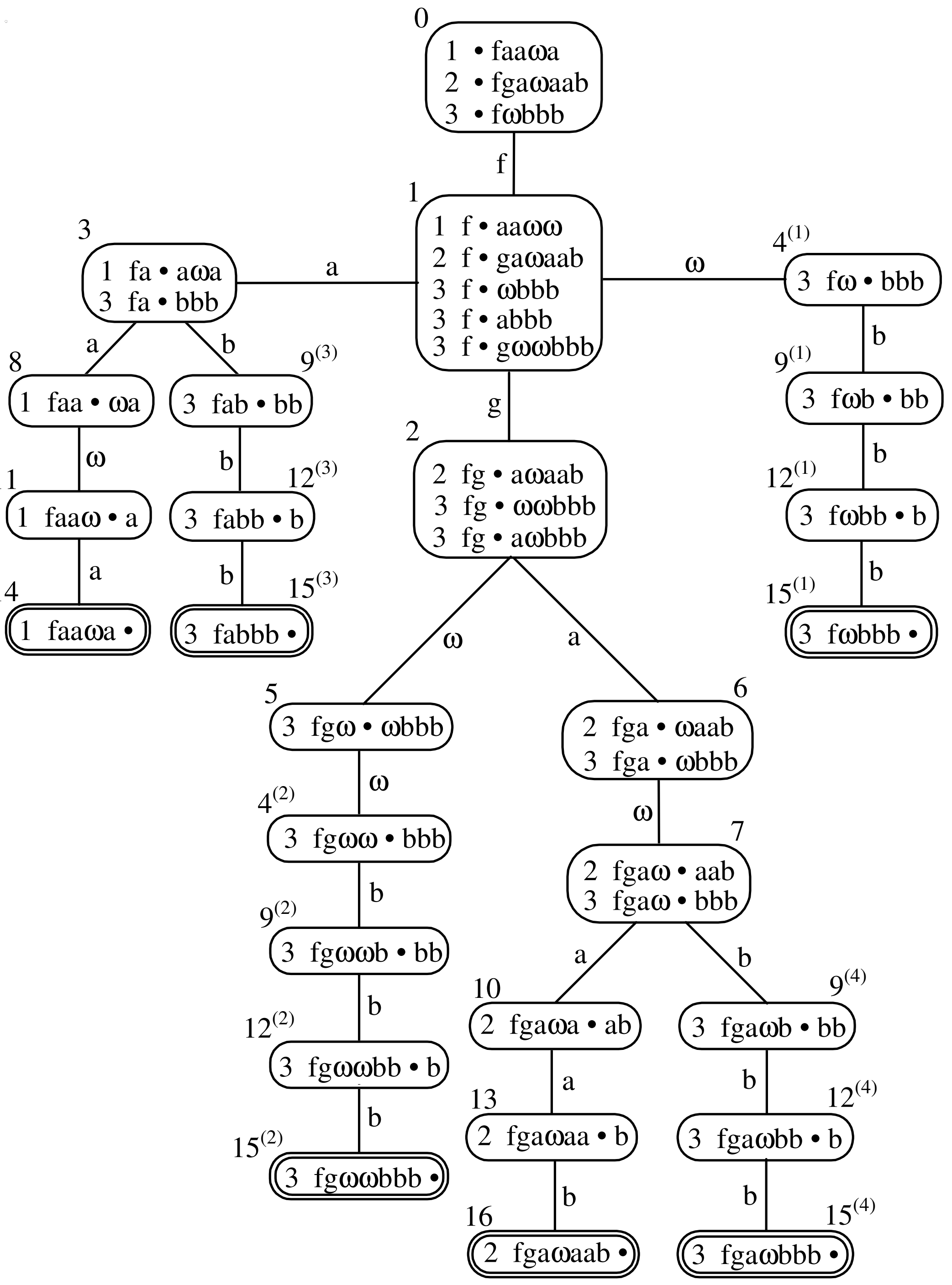

Figure 3.6: *Tree Automaton for* {1: *faaωa*, 2: *fgaωaab*, 3: *fωbbb*}

4. Optimal Pattern-Matching Automata

The tree automaton described above is time efficient during operation because it avoids symbol re-examination. However, it achieves this at the cost of increased space requirements. The unexpanded automaton corresponding to the pattern set of Figure 3.6, and to which no patterns are added, is given in Figure 4.1. It is much smaller. There, *fabbb* is only recognised by backtracking from state 2 to state 1 and then taking the branch through state 4 instead. But in Figure 3.6 a branch recognising *fabbb* has been added to avoid backtracking, thereby duplicating the existing sub-branch which recognises the *bbb* in *fωbbb*. We can see similar duplication in several other branches of Figure 3.6; those identified by sharing the same main state numbers.

By sharing duplicated branches, the tree automaton can be converted into an equivalent but smaller directed acyclic graph (*dag*) automaton. States which recognise the same inputs and assign the same rule numbers to them are functionally equivalent, and can be identified. For instance, the dag automaton corresponding to Figure 3.6 is given in Figure 4.2. The number of states is thereby reduced from 27 to 17, leaving just one state more than in the non-deterministic machine of Figure 4.1.

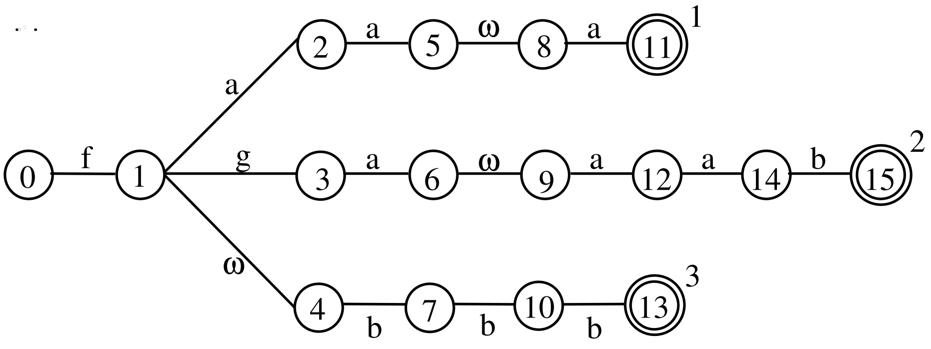

Figure 4.1: *Unexpanded Automaton for* { 1:*faaωa*,2:*fgaωaab*,3:*fωbbb*}

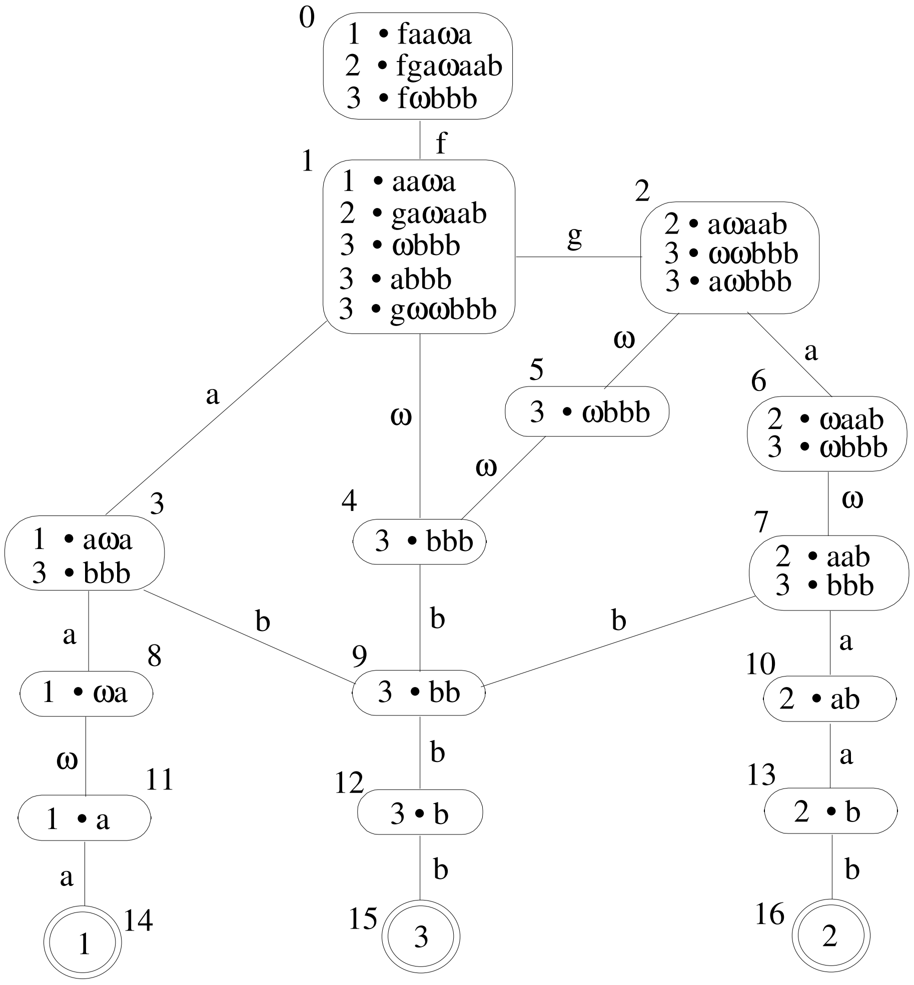

Figure 4.2: *Dag Automaton for* {1:*faaωa*, 2:*fgaωaab*, 3:*fωbbb*}

By sharing duplicated branches, the tree automaton can be converted into an equivalent but smaller directed acyclic graph (*dag*) automaton. States which recognise the same inputs and assign the same rule numbers to them are functionally equivalent, and can be identified. For instance, the dag automaton corresponding to Figure 3.6 is given in Figure 4.2. The number of states is thereby reduced from 27 to 17, leaving just one state more than in the non-deterministic machine of Figure 4.1.

The above example hides the complexity of recognising duplication where a number of suffixes are being recognised, not just one. The required dag automaton can be generated using finite state automaton minimisation techniques but this may require a lot of memory and time. The obvious alternative approach consists of using the matching sets to check new states for equality with existing ones while generating the automaton. In the case of equality, the new state is discarded and the existing one is shared. However, comparison of matching sets may be prohibitively expensive and it may well require bookkeeping for all previously generated matching sets. A major aim of this paper is to show how to avoid much of this work. First we must characterise states that would generate isomorphic subautomata.

Definition 4.3: Two matching items $r_1{:}\alpha_1{\bullet}\beta_1$ and $r_2{:}\alpha_2{\bullet}\beta_2$ are *equivalent* if, and only if, the suffixes and rule labels are equal, i.e. $\beta_1 = \beta_2$ and $r_1 = r_2$. Otherwise, they are *inequivalent*.

Definition 4.4: Two matching sets M_1 and M_2 are *equivalent* if, and only if, to every item i in $M_1 \cup M_2$ there correspond items $i_1 \in M_1$ and $i_2 \in M_2$ which are equivalent to i. Otherwise, the sets are *inequivalent*.

For instance, in Figure 3.6 the matching sets labelling the states $9^{(1)}$ and $9^{(2)}$ are equivalent whereas the matching sets labelling the states 3 and $4^{(2)}$ are inequivalent. Clearly, equivalence is the right criterion for coalescing nodes of the tree automaton to obtain the equivalent dag automaton: such sets will certainly accept the same pattern suffixes and result in the same rewrite rule being applied. So,

Lemma 4.5: Two matching sets generate identical automata if they are equivalent.

We believe this equivalence is actually necessary as well as sufficient to combine corresponding states in the automaton. However, equivalent matching sets may have different prefixes, as can be seen in Figure 3.6. Since only the suffixes are relevant to matching, only they appear labelling the states in Figure 4.2.

5. Dag Matching Automaton Construction

In this section, we describe how to build the minimised dag automaton efficiently without first constructing the tree automaton. This requires the construction of a list of matching sets in a suitable order to ensure that every possible state is obtained, and a means of identifying potentially equivalent states.

The items in matching sets all share a common prefix before the matching dot (e.g. see Figure 3.6), namely the string of symbols recognised before reaching the matching position. Hence, the matching position of any item is an invariant of the whole matching set. The states of the tree automaton can therefore be ordered using the left-to-right total ordering on the common matching positions of their matching sets. Unfortunately, states which are functionally equivalent may not share the same matching position (see [8]). So the matching position is not uniquely defined for a state in the dag automaton. However, one way of assigning an acceptable and unique matching position to a dag state is always to choose the leftmost (or the rightmost) position of all the states which have been coalesced. This is done starting with the initial state and working downwards.

The dag automaton is constructed as in Algorithm 5.1. We iteratively construct the machine using a list l of matching sets in which the sets are ordered according to the matching position of the set. So the initial matching set is first and final matching sets come last. Each set in l is paired with its corresponding state in the automaton and a pointer is kept to the current position in the list l.

Algorithm 5.1: Initialise l to contain just the pair of the initial matching set and state, and set the current pointer to it. For the iterative step, let $\langle M,S\rangle$ be the current pair in l. For each symbol $s \in F\cup\{\omega\}$, compute the non-empty matching sets $\delta(M,s)$ as defined above. If there is no pair $\langle M',S'\rangle$ in l such that $\delta(M,s)$ is equivalent to M' then create a state S', add it to the automaton with a transition labelled s from S to S' and insert the pair $\langle\delta(M,s),S'\rangle$ into l according to the matching position of $\delta(M,s)$. Otherwise, i.e. when such a pair $\langle M',S'\rangle$ already exists in l, create a transition s from S to S'. Lastly, increment the pointer to the next matching set in l, if such exists, and repeat. The process halts when the end of the list l is reached.

The list l represents the equivalence classes of matching states where the assigned matching position is that of the representative which is generated first. In each new set which is generated, the position of the current matching set is incremented at least one place to the right. So new members of l are always inserted to the right of the current position. This ensures that all necessary transitions will eventually be generated without moving the pointer backwards in l. It is easy to see from the definition of the *close* function that added patterns cannot contain positions that were not in one of the original patterns. So l contains sets with matching positions from a finite collection and, as each set can only generate a finite number of next states all of which are to the right, the total length of l is bounded and the algorithm must terminate.

The tree and dag automata clearly accept the same language and the automaton is minimal, in the sense that, by construction, none of the matching sets labelling the states in the automaton are equivalent. We now illustrate the algorithm.

Example 5.2: When applied to the pattern set of Example 3.5, the algorithm generates the same matching sets as in Figure 3.6, and we number the sets as there. It is clear from that figure that matching sets which are equivalent have been given the same number (with different bracketed subscripts for different occurrences) whereas inequivalent sets have different numbers. The left-to-right order by the matching position is given in Figure 5.3.

Matching position	List of states
Λ	0
1	1
1.1	2
1.2	5, 6
2	3, 4, 7
3	8, 9, 10
4	11, 12, 13
∞	14, 15, 16

Figure 5.3: *States of Figure 3.6 with their Matching Positions.*

One possible final list l which preserves this order is therefore given by reading it from top to bottom: (0, 1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16). As each state in this list becomes the current state, it is readily seen that the new states it generates (as given by the transitions from the current state in Figure 3.6) are further along the list because their matching positions are further to the right. In fact, the states are numbered in order of creation in this list: State 0 generates state 1 which in turn generates states 2, 3 and 4. Then state 2 generates states 5 and 6 which are to the left of 3 and 4 and so are inserted before them. Next, state 5 generates a state which is found to be equivalent to 4, so that in the dag there will be a transition to state 4 from both states 1 and 5. The process continues until the final states 14, 15 and 16 are reached (one for each initial pattern) when the algorithm terminates and the dag automaton of Figure 4.2 is obtained.

6. Checking for Equivalence

In this section, we show how matching sets can frequently be discriminated easily so that the cost of checking for equivalence is reduced. However, in some cases, comparison of suffixes in the matching sets cannot be avoided. We look at three properties to help achieve this. One is the set of rules represented by patterns in the matching set, another is the matching position and the third is a state *weight* function *wt*. The first and last yield the same values for equivalent states. We must extend the arity notation $\#f$ to include variable symbols and ω which are all considered to be like constants and so of arity 0, i.e., $\#\omega = 0$.

Definition 6.1: The weight *wt* of a string of function and variable symbols s_i is:

$$wt(s_1 \ldots s_n) \;=\; 1+\textstyle\sum_{i=1}^{n}(\# s_i - 1) \qquad \text{for } n \geq 0$$

and the weight of a matching item $\alpha{\bullet}\beta$ is defined as the weight of its prefix, viz.

$$wt(\alpha{\bullet}\beta) \;=\; wt(\alpha).$$

The following properties are readily verified (e.g. part (iii) by structural induction):

Lemma 6.2:

i) $wt(\varepsilon)$	=	1	i.e. the weight of the empty string is 1.
ii) $wt(\alpha_1\alpha_2 \ldots \alpha_n)$	=	$1+\sum_{i=1}^{n}(wt(\alpha_i)-1)$	for any n strings $\alpha_1, \alpha_2, \ldots, \alpha_n$.
iii) $wt(t)$	=	0	for any term t.
iv) $wt(\alpha{\bullet}\beta)$	=	n	if $\beta = t_1 \ldots t_n$ is a string of n terms.
v) $wt(\sigma(\alpha))$	=	$wt(\alpha)$	for any string α and substitution σ.

Using the arities as given previously to our example function symbols, we have for example:

$$wt(fag{\bullet}\omega aab) \quad = 1 + (\#f-1) + (\#a-1) + (\#g-1) \;=\; 1 + 3 - 1 + 1 \;=\; 4$$

$$wt(fafab{\bullet}\omega aab) \quad = 1 + (\#f-1) + (\#a-1) + (\#f-1) + (\#a-1) + (\#b-1) \;=\; 4$$

So the weight obtained is indeed the number of individual subterms after the matching dot. As symbol strings or as sequences of terms, these suffixes are identical although their parent patterns have different structures. For a matching item $\alpha{\bullet}\beta$, β is always a sequence of terms and so the weight function represents the number of terms in the suffix that have not yet been checked. This is the number of disconnected subtrees left after deleting the prefix nodes from the parse tree of the original term.

Since the prefix is an invariant of a matching set M, so is the weight and we can safely write $wt(M)$ for the common weight of any item in M. By (iv), the weight is determined uniquely from any suffix. So, as equivalent sets have the same suffix sets, the weight of a matching set is an invariant of its equivalence class. Equivalent matching sets must also have the same subsets of rules represented in their patterns.

Lemma 6.3: Equivalent matching sets have the same weight and rule sets.

In Figure 4.2, all inequivalent matching sets are distinguished by the use of either the weight function or the rule set. Thus the criteria are useful in practice. However, they will clearly not be sufficient in general. In the opposite direction, it is also sometimes easy to establish equivalence. Combining these with the matching position, we have the following very useful result which enables the direct checking of equivalence to be avoided entirely in Example 3.5.

Theorem 6.4: Matching sets that share a common matching position, weight and rule set are equivalent.

Proof: It suffices show the kernels of the matching sets are equivalent, since then the function *close* will add equivalent items to both sets. Let M_1 and M_2 be two matching sets that share the same weight, rule set and common matching position p. Let $i_1 = r{:}\alpha_1{\bullet}\beta_1$ and $i_2 = r{:}\alpha_2{\bullet}\beta_2$ be any two items associated with the same rule in their respectively kernels. The definitions of *accept* and *close* guarantee that the suffixes consist of a suffix of the original pattern π_r of the rule preceded by a number of copies of ω. To identify this suffix, let p' be the maximal prefix of the position p corresponding to a symbol in π_r. This is either the whole of p or is the position of a variable symbol ω. Either way, substitutions made by *close* for variables before p' in either i_1 or i_2 have already been fully passed in the prefix, and no substitution has yet been made for any variable further on in π_r. So if β is the suffix of π_r that starts at p' then the items i_1 and i_2 must have the form $\alpha_1{\bullet}\omega^{n_1}\beta$ and $\alpha_2{\bullet}\omega^{n_2}\beta$ for some $n_1, n_2 \geq 0$. Since M_1 and M_2 have equal weight, we have $wt(\alpha_1) = wt(\alpha_2)$. So $n_1 = n_2$ by Lemma 6.2(iv) and i_1 and i_2 are equivalent. We conclude that M_1 and M_2 are equivalent.

Although matching sets that share these three properties are equivalent, the matching positions of equivalent matching sets are not necessarily identical. An example can be found elsewhere (see [8]). Finally, we observe what weights and matching positions are possible:

Theorem 6.5: The weight and matching position of any matching set are the same as those of some matching item consisting of an original pattern with a matching dot.

Proof: From the definition of *close*, every symbol in an added pattern has the same position as a symbol in some pattern of the generating set. When that position becomes the matching position in the two corresponding patterns, the resulting items will have the same weight. By induction, these positions and weights must occur in the original pattern set.

7. Complexity

In this last main section, we evaluate the space complexity of the dag automaton by giving an upper bound for its size in terms of the number of patterns and symbols in the original pattern set. The bound established considerably improves Gräf's bound [2]. The *height* of a tree or dag automaton is the maximum distance from its root to a final state. The *breadth* of a tree automaton is the number of its final states, which is the size of the closure $\overline{L}$. Gräf bounds the size of the tree automaton by showing that

$$height(A_{tree}) \leq \sum_{\pi \in L} |\pi| \quad \text{and} \quad breadth(A_{tree}) \leq \prod_{\pi \in L} (|\pi|+1)$$

and then using the fact that the automaton size is less than the product of its breadth and height. We need a generalisation of breadth which is applicable to the dag automata here. We choose a rightmost matching position for each dag state so that positions move rightwards along every path. The following definition depends on this choice and so is not an invariant:

Definition 7.1: The *breadth* of a dag automaton is the maximum number of matching sets having the same matching position.

This wider definition of breadth coincides with the one for tree automata because:

Lemma 7.2: The maximum number of matching sets with the same matching position in a *tree* automaton is always the number of final matching sets, i.e. its breadth as a tree.

Proof: Along any branch from initial to final state of the *tree* automaton, the matching positions are all distinct because each state has a position to the right of its parent. Hence, for every occurrence of a given position in the tree, there is at least one occurrence of a final position at the end of the branch. Thus, there are at least as many final states as states with any given matching position.

Lemma 7.3: If A_{dag} is the dag automaton corresponding to the tree automaton A_{tree} then

$$breadth(A_{dag}) \leq breadth(A_{tree}) \quad \text{and} \quad height(A_{dag}) = height(A_{tree}).$$

Proof: The breadth inequality arises because every state in the dag corresponds to some state in the tree automaton that has the same assigned position. Also, the height of the dag and tree automata must coincide since the former has no cycles and so paths in the tree cannot become shorter in the dag.

The size of the tree automaton is bounded above by the product of its height and breadth. The nearest equivalent result for dag automata is that its size is bounded by the product of the number of different matching positions and its breadth. In the following, the bound on the breadth of the dag automaton is much better than that above for the tree automaton, and the bound for the number of positions essentially duplicates the height bound for the tree automaton. So immediately we have a much better overall bound on the size of the dag automaton than that given by Gräf for the tree automaton.

Lemma 7.4: If at least one pattern has 2 or more symbols, then the breadth of the dag automaton for a pattern set L is bounded above by $(2^{|L|}-1)(Max_{\pi \in L}|\pi|-1)$ where $|\pi|$ is the length of pattern π and $|L|$ is the size (cardinality) of L.

Proof: There is only one state, the initial one, which has the initial position and there are exactly $|L|$ states which have the final position, one for each pattern of L. So the breadth bound holds for them. Otherwise assume that the breadth is determined by a non-initial, non-final position. By Theorem 6.4, the maximum number of inequivalent states with the same matching position is bounded by the product of the number of different possible weights and the number of different possible rule sets. The rule set for any state could be any non-empty subset of L, of which there are $2^{|L|}-1$. So it remains to show that $Max_{\pi\in L}|\pi|-1$ is an upper bound on the number of possible weights. Using Theorem 6.5, the weight of a state is the weight of a prefix of a rule in L. This is the number of terms that could appear in its suffix. As we ignore initial and final positions, the weight is bounded below by 1 and above by one less than the number of symbols in the suffix, and hence by the length of the pattern minus 1. So up to at most $Max_{\pi\in L}|\pi|-1$ different weights are possible.

Lemma 7.5: The height of the minimised dag for a pattern set L is bounded above by $2 - |L| + \sum_{\pi\in L}|\pi|$ and the number of its distinct positions is bounded above by $1 + \sum_{\pi\in L}|\pi|$.

Proof: Along a path from the initial node to a final node, the positions are all distinct, each child having a position to the right of its parent, with each position appearing in some pattern of L. The number of non-initial, non-final positions counting multiplicities is $\sum_{\pi\in L}(|\pi|-1)$. Adding $1+|L|$ to this gives an upper bound on the total number of distinct positions, whereas adding 2 gives an upper bound on the height of the dag.

Combining the methods of proof for these two results above yields a very much better bound on dag automaton size than simply taking the product of breadth and number of positions. Our main conclusion about the space efficiency of the dag automaton described here is the following:

Theorem 7.6: The size of the dag automaton for a pattern set L is bounded above by:

$$1 + |L| + (2^{|L|}-1)\left(\sum_{\pi\in L}(|\pi|-1) \right)$$

Proof: The number of states in the minimised dag automaton is $1+|L|$ plus the number of non-initial, non-final states in the dag. By Theorem 6.4, we just need to count the number of possible (*weight*, *position*) pairs, and multiply by the bound $2^{|L|}-1$ on the number of different possible rule sets. By Theorem 6.5, every (*weight*, *position*) pair of interest is given by a non-initial symbol position in an original pattern. So the number of such pairs is bounded by the number of such symbols, i.e. by $\sum_{\pi\in L}(|\pi|-1)$.

The above bound is easily computable with no internal knowledge of the patterns. However, if we have more specific information about subsets of rules starting with given function symbols, then the following improved bound may be applicable:

Corollary 7.7: The size of the dag automaton for a pattern set L is bounded above by:

$$1 + |L| + \sum_{f\in F^*} \{\ 1 + (2^{|L\backslash f\,|}-1)(\ \sum_{\pi\in L\backslash f}(|\pi|-1)\ )\ \} + \ \delta(\ Max_{\pi\in L}|\pi|-2\ )$$

where $L\backslash f$ is as in Definition 2.4, F^* is the subset of function symbols which appear as first symbols in L, and $\delta = 1$ if ω is a pattern of L and L has a pattern of length at least 3, and $\delta = 0$ otherwise.

Proof: Assume, first of all, that ω is not itself a pattern in L. So $L = \bigcup_{f\in F^*} fL \setminus f$ and $|L| = \sum_{f\in F} |\ L\backslash f\ |$. Then, after the initial state, the automaton splits into disjoint subautomata A_f which are entered according to the symbol f read first. Thus the size of the automaton is bounded by the number $1+|L|$ of initial and final states, plus 1 for the initial state of each A_f plus the number of non-initial, non-final states in each A_f. The rule set available for each state in A_f is restricted to a subset of those in $L\backslash f$ and the (*weight*, *position*) pairs of states must correspond to symbol positions in patterns of $L\backslash f$. So we can apply Theorem 7.6 to bound A_f in terms of $L\backslash f$ and so obtain the result. Now assume that ω is the initial symbol of a pattern. Then it is actually the whole pattern, so A_ω would consist of a single final state. The *close* function would add patterns to each A_f so that failure to match a pattern of $fL\backslash f$ would result in passing directly to a sequence of states which recognises all remaining input symbols and terminates at the final state of A_ω. For convenience, we will count this sequence of states as part of A_ω. Then, apart from extra transitions, the only difference the pattern ω makes is to add A_ω, which has at most $Max_{\pi\in L}|\pi|-2$ non-final states.

8. Conclusion

First, we described a practical method that compiles a set of prioritised overlapping patterns into an equivalent deterministic automaton which does not need backtracking to announce a match. Re-examination of symbols while matching terms is completely avoided. The matching automaton can be used to drive the pattern-matching process with any rewriting strategy.

In the main body of the paper, we described a method to generate an equivalent minimised dag matching automaton very efficiently without constructing the tree automaton first. We directly built the dag-based automaton by identifying the states of the tree-based automaton that would generate identical subautomata. By using the dag-based automata we can implement left-to-right pattern-matchers that avoid symbol re-examination without much increase in the space requirements.

Some useful functions were described for distinguishing inequivalent states when building the dag automaton. A theorem which guaranteed equivalence in terms of several simple criteria was then applied to established improved upper bounds on the size of the dag automaton in terms of just the number of patterns and symbols in the original pattern set. These considerably improve Gräf's previous bounds for the tree automaton.

References

[1] Aho, A.V., Sethi, R. and Ulmann, J.D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, 1986.

[2] Gräf, A., “Left-to-Right Pattern-Matching”, in Proc. *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Vol. 488, pp. 323-334, Springer Verlag, 1991.

[3] Hoffman, C.M. and O’Donnell, M.J., “Pattern-Matching in Trees”, *Journal of the ACM*, Vol 29, pp. 68-95, January 1982.

[4] Hoffman, C.M., O’Donnell, M.J. and Strandth, R.I., “Programming with Equations”, *Software, Practice and Experience*, Vol. 15, No 12, pp. 1185-1204, December 1985.

[5] Johnson, S. C., *Yacc - Yet Another Compiler Compiler*, Computing Science Technical Report 32, AT&T Laboratories, Murray Hill, N. J., 1975.

[6] Kennaway, J. R., “The Specificity Rule for Lazy Pattern Matching in Ambiguous Term Rewriting Systems”, *Proc. 3rd European Symposium on Programming*, Lecture Notes in Computer Science, Vol. 432, pp. 256-270, Springer -Verlag, 1990.

[7] Knuth, D.E., Morris, J. and Pratt, V., “Fast Pattern-Matching in Strings”, *SIAM Journal*, Vol. 6, No. 2, pp. 323-350, 1977.

[8] Nedjah, *Pattern-Matching Automata for Efficient Evaluation in Equational Programming*, PhD Thesis, UMIST, Manchester, UK, 1997.

[9] Nedjah, N., Walter, C.D. and Eldridge, S.E., *Efficient Automaton-Driven Pattern-Matching for Equational Programs*, Technical Report, Computation Dept., UMIST, Manchester, UK, 1996.

[10] O’Donnell, M.J., *Equational Logic as a Programming Language*, The MIT Press, 1985.

[11] Sekar, R.C., Ramesh, R. and Ramakrishnan, I.V., “Adaptive Pattern Matching”, *SIAM Journal of Computing*, Vol. 24, No. 5, pp. 1207-1234, December 1995.