More Efficient Pattern-Matching Automata for Overlapping Patterns

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

Pattern-matching is a fundamental feature in many applications such as functional programming, logic programming, term rewriting and rule-based expert systems. Usually, patterns are pre-processed into a deterministic finite automaton. Using such an automaton allows one to determine the matched pattern(s) by a single scan of the input term. The matching automaton is typically based on left-to-right traversal of patterns. With overlapping patterns a subject term may be an instance of more than one pattern. To select the pattern to use, a priority rule is usually engaged.

The pre-processing of the patterns adds new patterns which are instances of the original ones. When the original patterns overlap, some of the instances supplied may be irrelevant for the matching process. They may cause an unnecessary increase in the space requirements of the automaton and may also reduce the time efficiency of the matching process. Here, we devise a new pre-processing operation that recognises and avoids such irrelevant instances and hence improves space and time requirements for the matching automaton.

Keywords: Pattern-matching, tree automaton, closure, compilation.

1. Introduction

Pattern-matching is an important operation in several applications such as functional, equational and logic programming [5,11], theorem proving [4] and rule-based expert systems [3]. It can be achieved as in lexical analysis by using a finite automaton [2,6,7,10,12]. In order to avoid backtracking over symbols already examined, new patterns are added. These correspond to overlaps in the scanned prefixes of original patterns. When patterns overlap, some of the added patterns may be irrelevant to the matching process. Sekar et al. [12] use the notion of *representative* pattern sets to compute smaller automata for pattern-matching by deleting such patterns. As each automaton node is constructed, their algorithm eliminates a pattern π from the current matching set whenever a match for π implies a match for a pattern of higher priority. This deletion is a computationally expensive operation and loses some potentially useful feed-back to the programmer. In this paper, we focus on improving these aspects by leaving the

---

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

removal of irrelevant patterns till the end of the construction. This usually results in more efficient compilation.

First, we recall from [10] a method for generating a deterministic tree matching automaton for a given pattern set. Although the generated automaton is efficient since it avoids symbol re-examination, it may contain unnecessary branches. Part of this is due to duplication of equivalent nodes. [10] provides an efficient method for recognising most of these and so obtains an upper bound for the size of an equivalent, partially minimised automaton which improves on the bounds in [6] and [12]. The other main reason for excessive automaton size is the presence, or introduction during the construction process, of overlapping patterns which are not eventually matched. Here, a smaller automaton may be obtained by pruning back from final nodes.

2. Notation

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 $ft_1t_2...t_{\#f}$ where each t_i, $1 \leq i \leq \#f$, is itself a term. We abbreviate terms by removing the usual parentheses and commas. This is unambiguous in our examples since the function arities will be kept unchanged throughout, namely $\#f = 3$, $\#g = 1$, $\#a = \#b = 0$. A term containing no variables is said to be a *ground* term. Throughout, variable occurrences are replaced by ω since the actual symbols are irrelevant here. (Often _ is used instead.) The justification for this is that we assume all patterns are linear terms, i.e. each variable symbol can occur at most once in each. Pattern sets will be denoted by L and patterns by π_1, π_2, ..., or simply by π. A term t is said to be an *instance* of a (linear) pattern π if t can be obtained from π by replacing the variables of π by corresponding subterms of t.

Definition 2.1: A *matching item* is a triple $r{:}\alpha{\bullet}\beta$ where $\alpha\beta$ is a term and r is a *rule label* identifying a pattern in the original, prioritised pattern set. 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*, and α and β are called the *prefix* and *suffix* respectively. A *final* matching item is one of the form $\alpha\bullet$.

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 2.2: 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.

Definition 2.3: 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 and excluding the other members of L. Then, for $f \in F$ define L_ω and L_f by:

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

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

where $\omega^{\#f}$ denotes a string of $\#f$ symbols ω and $\emptyset$ is the empty set. The *closure* $\overline{L}$ of a pattern set L is then defined recursively by Gräf [6] as follows:

$$\overline{L} = \begin{cases} L & \text{if } L = \{\varepsilon\} \text{ or } L = \emptyset \\ \bigcup\limits_{s \in F \cup \{\omega\}} s\,\overline{L_s} & \text{otherwise} \end{cases}$$

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 briefly recall from [10] a practical method to construct a deterministic tree matching automaton for a prioritised overlapping pattern set. The pattern set L is extended to its closure $\overline{L}$ while generating the matching automaton.

The automaton 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 the initial state contains the initial matching items formed from the original patterns and labelled by the rules associated with them. The state transition function is denoted by δ. For each symbol $s \in F \cup \{\omega\}$ and state S with matching set M, the new state $\delta(S,s)$ has matching set $\delta(M,s)$ derived by applying the composition of the functions *accept* and *close* defined in Figure 3.1.

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

Figure 3.1: *Automata Transition Function from [10].*

The items obtained by recognising the symbols in those patterns of M where s is the next symbol form the set $accept(M,s)$. 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 $r{:}\alpha{\bullet}f\omega^{\#f}\mu$ is added to remove this non-determinism and avoid backtracking. Its label simply keeps account of the originating pattern from which it was derived. Now the f-transition can be taken when f is the next input symbol regardless of which of the two patterns, if any, is matched. The transition function δ thus implements simply the main step in the closure operation described by Gräf [6] and set out in the previous section. Hence the full pattern set resulting from all additions during the automaton construction coincides with the closure operation of Definition 2.3.

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 distinct overlapping patterns then either π_1 has *higher* priority than π_2 or π_2 has *higher* priority than π_1. In the latter case, we write $\pi_1 < \pi_2$.

Since the rule labels of items identify patterns from a prioritised pattern set, we can write $r_1 < r_2$ for labels r_1 and r_2. When a final state is reached, if several patterns have been successfully matched, then the priority rule is engaged to select the one of highest priority. An example is the *textual* priority rule which is used in the majority of functional languages [1,8,9,13]. Among the matched patterns, the rule chooses the pattern that appears first in the text. Whatever priority rule is used, we will apply the word *match* only to the pattern of highest priority which is matched:

Definition 3.4: For a prioritised pattern set L and pattern $\pi \in L$, the term t is said to *match* π *in* L if, and only if, t is an instance of π but not an instance of any other pattern in L of higher priority than π.

Example 3.5: Let $L = \{1{:}fa\omega\omega,\ 2{:}f\omega aa,\ 3{:}f\omega ba,\ 4{:}fg\omega g\omega b\}$ be the pattern set where $\#f = 3$, $\#g = 1$ and $\#a = \#b = 0$, as throughout this paper. Assuming a textual priority rule, the matching automaton for L is given in Figure 3.6. Transitions corresponding to failures are omitted. Each state is labelled with its matching set. In the construction process, each new item is associated with the rule from which it is directly derived and whose pattern it is known to match. So, an added item $\alpha\bullet f\omega^{\#f}\beta$ is associated with the same rule as is its parent $\alpha\bullet\omega\beta$. Items added by *close* are underneath a horizontal line in the state. At the final nodes, several items may be matched. Then the highest priority matching rule must be chosen, but the others appear in parentheses. When this happens, it indicates the occurrence of what we call *irrelevancy* in the next section. During pattern matching, an ω-transition is only taken when there is no other available transition which accepts the current symbol; these transitions might more aptly be labelled "*else*" when there are other alternatives starting at the parent node. With care, the automaton can usually be used to drive the pattern-matching process irrespective of the chosen term rewriting strategy.

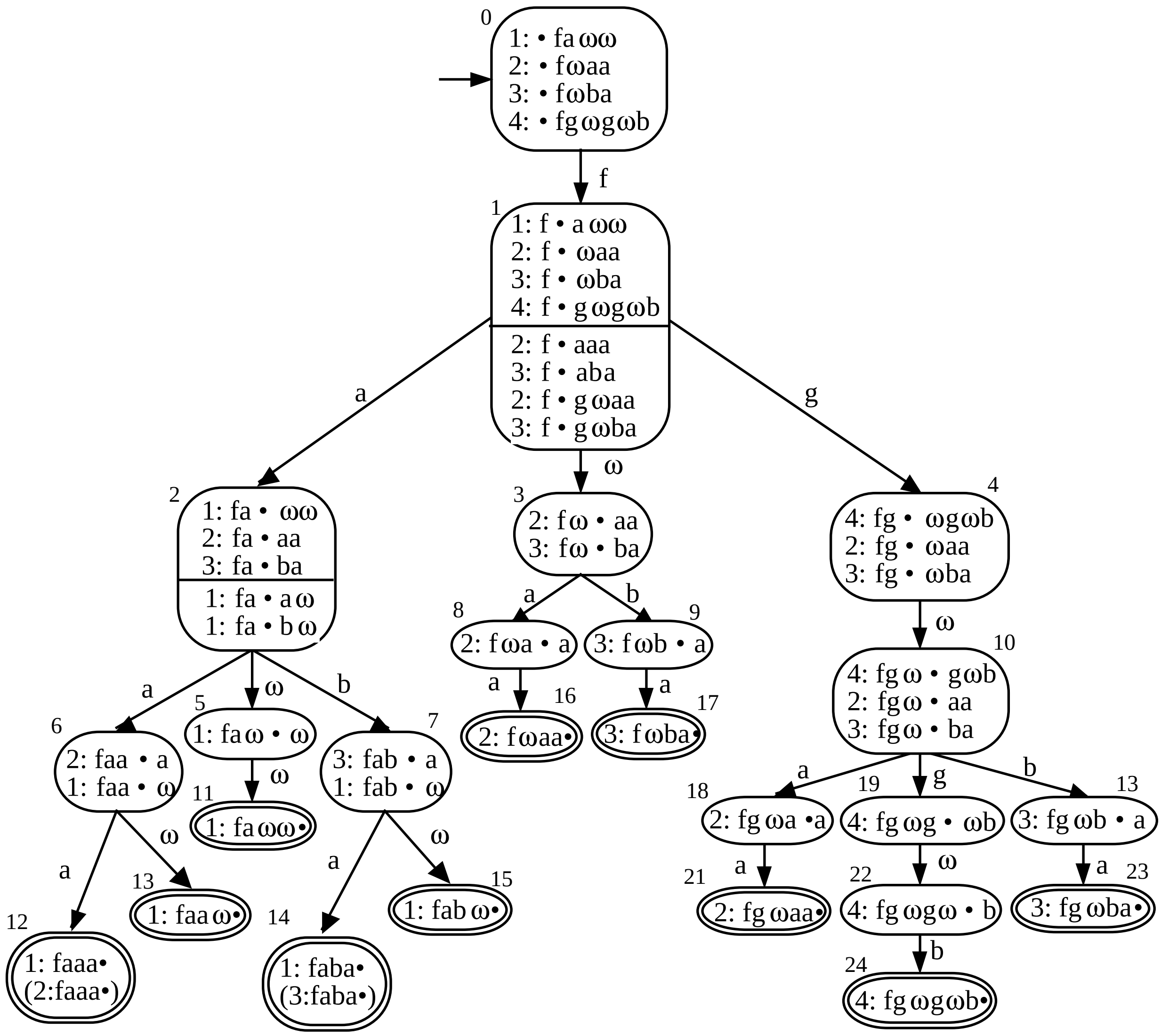

Figure 3.6: *Matching Automaton for {1:faωω, 2:fωaa, 3:fωba, 4:fgωgωb}.*

Proc. Internat. Workshop on the Implementation of Functional Languages, St Andrews, Sept 1997, pp 341-350

4. Reduced Matching Automaton for Overlapping Patterns

We now treat the observation that the *close* function of Figure 3.1 may add more items than it needs when the patterns are overlapping. For example, in Figure 3.6 consider the items 2:*f•aaa* and 3:*f•aba* which *close* adds to state 1 but which are finally ignored in states 12 and 14. Every term matching these items is eventually associated with derivatives of the higher priority item 1:$f \bullet a\omega\omega$, also of state 1, and so the two items could have safely been omitted. We return to this example again after several definitions.

A *position* in a term is a path specification which identifies a node in the parse tree of the term. For a term t and a position p in this term, we denote by $t[p]$ the symbol at position p in t.

Definition 4.1: A term t is said to be *more general than* (*a term*) t' *at* (*the position*) p if, and only if, the symbol $t[p]$ is ω, $t'[p]$ is a function symbol and the prefixes of t and t' ending immediately before p are the same. We will also say t is *initially more general than* t' if t is more general than t' at the first position for which the symbols of t and t' differ.

It is the existence of a pair of terms with one being more general than the other at the matching position that is the condition for the function *close* to add in a new item. The priority rule of the original rule set must be extended to these new items. The following definition enables us to associate a unique item of highest priority from amongst all those which pattern-match a given term: if there is a unique pattern-matched item whose rule has highest priority then that is chosen; otherwise, when there are several pattern-matched items associated with rules of maximal priority, it turns out that those items must all derive from the same original pattern and we can select the initially most general. (Any other uniquely defined choice would also be acceptable, but this is the most convenient in what follows.)

Definition 4.2: Item $r{:}\alpha \bullet \beta$ has *higher priority* than item $r'{:}\alpha' \bullet \beta'$ if r has higher priority than r' or $r = r'$ and $\alpha\beta$ is initially more general than $\alpha'\beta'$. For a matching set M and item $r{:}\alpha \bullet \beta \in M$, the term t is said to *match* $r{:}\alpha \bullet \beta$ *in* M if, and only if, t is an instance of $\alpha\beta$ but not an instance of any other item in M of higher priority.

In fact, for a ground term there will always be a unique choice for the (highest priority) matching item because each matching set in the automaton will contain a matching instance of the unique original pattern which the priority rule identifies as *the* match to choose for the input term. Note that although the pattern $\alpha\beta$ of the item $r{:}\alpha \bullet \beta$ will always be an instance of the pattern of rule r, instances of it may actually match a pattern of higher priority. Such a higher priority rule could not have been used in generalising the priority rule to all terms and items because it is not uniquely defined; different instances of a non-ground term may match different patterns. The above definition suffices because for any given input term, each matching set will contain a matching item associated with the correct rule of maximal priority. It is now possible to determine which patterns are useful for matching sets to include. Indeed, we can start by considering the usefulness of each pattern in the initial pattern set:

Definition 4.3: Suppose $L\cup\{\pi\}$ is a prioritised pattern set. Then π is said to be *relevant for L* if there is a term that matches π in $L\cup\{\pi\}$ in the sense of Definition 3.4. Otherwise, π is *irrelevant for L*. Similarly, an item π is *relevant for (the matching set) M* if there is a term that matches π in $M\cup\{\pi\}$ in the sense of Definition 4.2.

Clearly, any term that matches an element of a pattern set, respectively item of a matching set, will still have that property even when an irrelevant pattern, respectively item, is removed. We can therefore immediately prune irrelevant patterns one by one from the initial pattern set until every remaining pattern is relevant to the remaining pattern set, and do the same for each matching subset generated by *accept*. Sekar et al. [12] call the resulting sets *representative* sets.

As observed above, the function *close* of Figure 3.1 may certainly supply items that are irrelevant for subsequent matching. Indeed, *all* additional items which *close* supplies are irrelevant for the current state although they may be relevant to later matching sets; nevertheless we need them for the child states to avoid backtracking. The uselessness of some items for subsequent matching may happen when the original pattern set contains overlapping patterns with more general ones having lower priorities. For instance, in Example 3.5, the original patterns 1:$fa\omega\omega$ and 2:$f\omega aa$ are overlapping, 2:$f\omega aa$ is more general than 1:$fa\omega\omega$ at position 1 yet 1:$fa\omega\omega$ has higher priority. The *close* function supplies the items 2:$f{\bullet}aaa$, 3:$f{\bullet}aba$ and two others to state 1. Then accepting symbol a would yield a superset of $\{1{:}fa{\bullet}\omega\omega, 2{:}fa{\bullet}aa, 3{:}fa{\bullet}ba\}$. At this stage, based only on the item 1:$fa{\bullet}\omega\omega$ a match for $fa\omega\omega$ can be announced and hence items 2:$fa{\bullet}aa$ and 3:$fa{\bullet}ba$ are redundant, and indeed irrelevant under the definition above.

In order to avoid irrelevant items, we can follow Sekar [12] and define a new function *rel* which, given a matching set M, removes all the irrelevant items. A definition of this function is given in Figure 4.4. For a matching set M and a symbol $s \in F\cup\{\omega\}$, the automaton transition function is now defined by $\delta(M,s) = close(rel(accept(M,s)))$. This removes the items corresponding to rules 2 and 3 from state 2 in Figure 3.6 and yields the reduced automaton of Figure 4.6.

$rel(M)$	$=$	any maximal subset S of M such that
		if $\pi \in M\backslash S$ then π is irrelevant for S.

Figure 4.4: *Rel function which deletes irrelevant items.*

Checking for relevance is mostly straightforward. Consider items in descending order of priority. Suppose $r{:}\alpha{\bullet}\beta$ is being considered for relevance in $M = accept(M_0, s)$. Any item $r'{:}\alpha'{\bullet}\beta' \in M$ may be ignored as far as $r{:}\alpha{\bullet}\beta$ is concerned if the priority rules do not relate r and r'. This is because $\alpha{\bullet}\beta$ and $\alpha'{\bullet}\beta'$ are special cases of the patterns r and r' respectively, which cannot overlap. So, such $r'{:}\alpha'{\bullet}\beta'$ do not affect the relevance of $r{:}\alpha{\bullet}\beta$. At the opposite extreme, suppose there is an item $r':\alpha'\bullet\beta' \in M$ for which $r = r'$. Then both items are instances of the pattern of rule r with prefix $\alpha = \alpha'$. Both β and β' must be the complementary suffix of the pattern of rule r preceded by the same number of ωs (see [10]). So, the items are identical and $r:\alpha\bullet\beta$ is irrelevant. Indeed, this argument shows that *rel* will produce a set in which every item is associated with a different rule. This leaves the situation where all items $r':\alpha'\bullet\beta' \in M$ affecting the relevance of $r:\alpha\bullet\beta$ satisfy $r < r'$. All items have the same prefix α followed by some inserted occurrences of ω, followed by a suffix of the named rule. Hence, the most general unifier of $r:\alpha\bullet\beta$ and any $r':\alpha'\bullet\beta'$ can be found easily from that of r and r', which can be calculated initially once for all. Standard techniques (see [12]) enable the relevance or not of $r:\alpha\bullet\beta$ to be concluded. However, we feel it worth noting that in a real application, especially in a typed system, it is often the case that the number of function symbols which can be used at a given position in a term is finite. This means that it may be possible to cover an item $r:\alpha\bullet\beta$ which contains ω at some position with items of higher priority containing a function symbol at that position. Then, contrary to some algorithms, the ω cannot be instantiated as some unseen function symbol to show the item is relevant.

Example 4.5: Using the improved transition function, the automaton corresponding to $L = \{1{:}fa\omega\omega,\ 2{:}f\omega aa,\ 3{:}f\omega ba,\ 4{:}fg\omega g\omega b\}$ is given in Figure 4.6. As usual, transitions corresponding to failure are omitted. Notice that for the same pattern set the automaton of Figure 3.6 has six more states, namely states 6, 7, 12, 13, 14 and 15. Pattern-matching for the terms *faaa* and *faba* using the automaton of Figure 3.6 necessitates four symbol examinations whereas by using the automaton of Figure 4.6 only two symbols need to be examined as ωs match any term. Thus, using the new transition function in this example, not only does the automaton have fewer states but it also allows pattern-matching to be performed more quickly for some inputs.

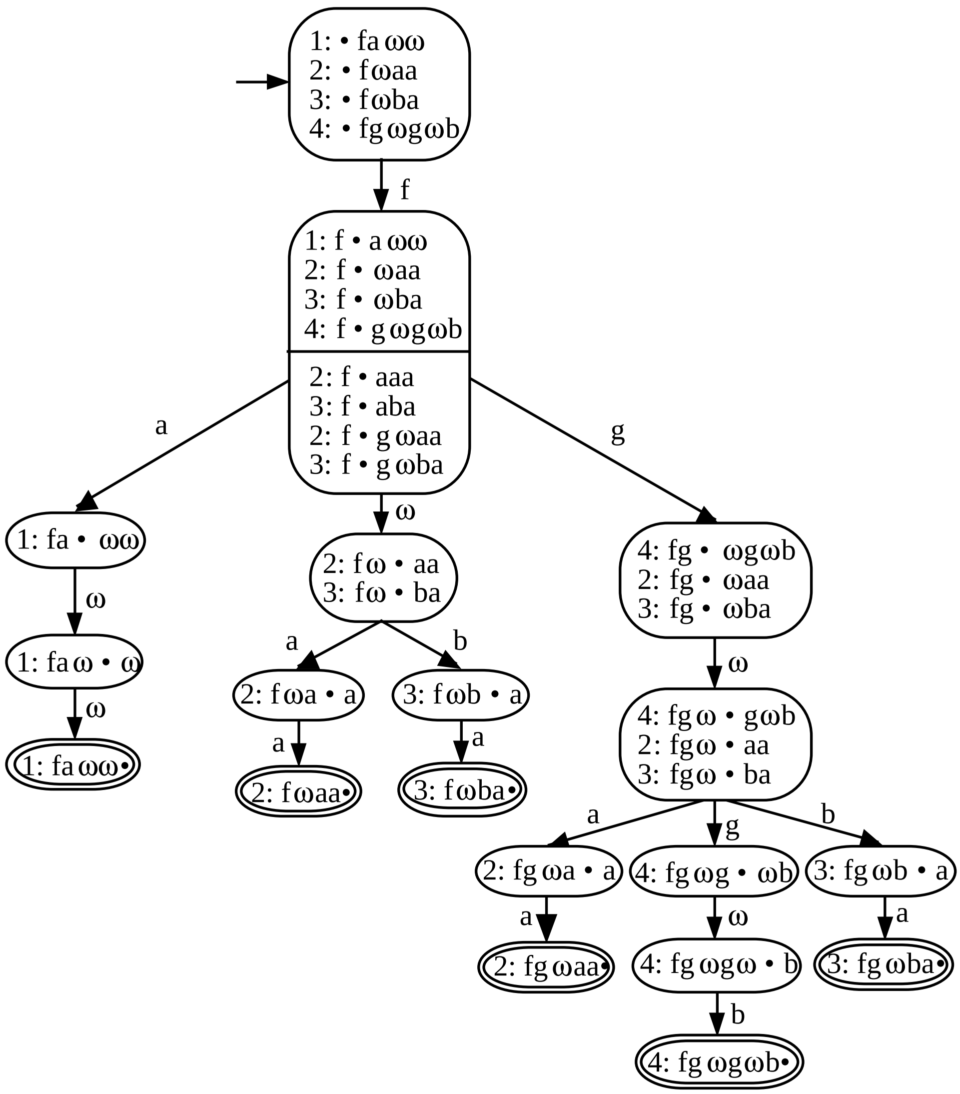

Figure 4.6: *Matching automaton using the function* rel.

In practice it may be expensive to check relevance as every state is constructed. It may be cheaper to build the full automaton first and cut back the branches to remove irrelevancy afterwards. The algorithm for this is the following. Items are first removed from final nodes, so that each final node contains a single item, naming the matching rule to be applied. Then, moving up the matching tree, whenever an item has had all its images under *accept* removed from child nodes, it can be removed itself from the current node and *close* re-applied to modify the children if necessary. Thus, in Figure 3.6 the final states 12 and 14 both contain an irrelevant pattern for deletion: only the rule numbers differ between the items and we must select that of highest priority and discard the other. After this the items corresponding to rules 2 and 3 may be deleted from states 6 and 7 and hence from state 2 because they are not used in any of their child states. Re-computing the closure of state 2 causes the branches rooted at states 6 and 7 to disappear. The automaton of Figure 4.6 is thus obtained.

To justify the correctness of this algorithm, note first that an irrelevant item, say $r{:}\alpha{\bullet}\beta$, cannot be matched. At the final node reached by inputting the irrelevant item $\alpha\beta$, by construction there must be the instance $\alpha\beta$ of *every* original pattern r' which matches $\alpha\beta$, annotated with r'. These are all irrelevant except for the item associated with the unique rule of highest priority, Thus the automaton contains irrelevant items if, and only if, there is a final node whose matching set contains more than one item. Now suppose that all occurrences of an item $r{:}\alpha{\bullet}\beta \in M$ have been removed from the children of the node labelled by M. We need to establish the correctness of removing it from M also. First of all, by induction, the item must be irrelevant for M: if it is irrelevant for, or unused by, all the children, then no input reaching M will match the item (else the item would appear in a final node descendent, which it doesn't), so it is irrelevant. The other question is whether removal affects the pattern-matching. However, we need not pattern match an irrelevant item, since all inputs to the node are matched with other items. Lastly, is the item necessary to avoid backtracking? Here we note that if the item is to be removed from the *kernel* of the matching set, that is, the set obtained by applying *accept* to the parent set, then *close* is re-applied to the new set so that backtracking is automatically avoided. However, if $r{:}\alpha{\bullet}\beta$ is itself added by *close*, then it can be safely deleted without re-applying *close* because, as the item will not be matched, there is no need to back-track to it. Thus the pruning algorithm is correct.

This process can be applied equally well to the partially minimised acyclic automata described by the authors in [10], and enables a similar reduction in automaton size there also.

Finally, we observe the usefulness of this approach to the programmer. The appearance of irrelevant items in final states corresponds to overlapping patterns; the automaton construction process identifies overlaps automatically so that, without extra effort, the programmer can be advised if appropriate.

8. Conclusion

In this paper, we began by recalling a practical method that compiles prioritised overlapping patterns to a deterministic matching automaton. However, the automaton may include unnecessary branches when the patterns are overlapping. In the main body of the paper, we identified those branches by considering relevant patterns and, by deleting irrelevant patterns, we modified the method so that only necessary branches are included in the matching automaton. An example showed how the matching automaton obtained can improve both the space and time requirements.

References

[1] A. Augustsson, *A Compiler for Lazy ML*, Proc. ACM Conference on Lisp and Functional Programming, ACM, pp. 218-227, 1984.

[2] J. Christian, *Flatterms, Discrimination Nets and Fast Term Rewriting*, Journal of Automated Reasoning, vol. **10**, pp. 95-113, 1993.

[3] D. Cooper and N. Wogrin, *Rule-Based Programming with OPS5*, Morgan Kaufmann, San Francisco, 1988.

[4] N. Dershowitz and J.P. Jouannaud, *Rewrite Systems*, Handbook of Theoretical Computer Science, vol. 2, chap. 6, Elsevier Science Publishers, 1990.

[5] A.J. Field and P.G. Harrison, *Functional Programming*, International Computer Science Series, Addison-Wesley, 1988.

[6] A. Gräf, *Left-to-Right Tree Pattern-Matching*, Proc. Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science vol. **488**, pp. 323-334, Springer-Verlag 1991.

[7] C.M. Hoffman and M.J. O’Donnell, *Pattern-Matching in Trees*, Journal of ACM, vol. **29**, nº 1, pp. 68-95, 1982.

[8] P. Hudak, S.L. Peyton-Jones, P. Wadler, et al., *Report on the Programming Language Haskell: a Non-Strict, Purely Functional Language, Version 1.2*, SIGPLAN Notices, Section S, May 1992, pp. 1-164.

[9] A. Laville, *Comparison of Priority Rules in Pattern Matching and Term Rewriting*, Journal of Symbolic Computation, nº **11**, pp. 321-347, 1991.

[10] N. Nedjah, C.D. Walter and S.E. Eldridge, *Optimal Left-to-Right Pattern-Matching Automata*, Proc. ALP’97, Southampton, Lecture Notes in Computer Science vol. **1298**, Springer-Verlag, 1997.

[11] M.J. O'Donnell, *Equational Logic as Programming Language*, MIT Press, 1985.

[12] R.C. Sekar, R. Ramesh and I.V. Ramakrishnan, *Adaptive Pattern-Matching*, SIAM Journal on Computing,, vol. **24**, nº 5, pp. 1207-1234, 1995.

[13] D.A. Turner, *Miranda: a Non Strict Functional Language with Polymorphic Types*, Proc. Conference on Lisp and Functional Languages, ACM, pp. 1-16, 1985.