SOFTWARE–PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(9), 793-813 (1999)

Efficient Automata Driven Pattern Matching for Equational Programs

NADIA NEDJAH[*¶], COLIN D. WALTER AND STEPHEN E. ELDRIDGE
Computation Department, UMIST, PO Box 88, Manchester M60 1QD, UK
(*email*: {*nn, cdw, see*}*@sna.co.umist.ac.uk*)

SUMMARY

We propose a practical technique to compile left-to-right pattern-matching of prioritised overlapping function definitions in equational languages to a matching automaton from which efficient code can be derived. First, a matching table is constructed using a compilation method similar to the technique that YACC employs to generate parsing tables. The matching table obtained allows for the pattern-matching process to be performed without any backtracking. Then, the known information about right sides of the equations is inserted in the matching table in order to speed-up the pattern-matching process. Most of the discussion assumes that the processed pattern set is left-linear, the non-linear case being handled by an additional pass following the matching stage. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: compilation; equational programming; pattern-matching; rewriting; reduction strategy

1. INTRODUCTION

As research has proved so far, the pattern-matching feature of equational and functional languages is very expressive, fully compensating for the lack of side effects in such languages.

The operational semantics behind the rewriting-based programming paradigm is now well-known [1–3]. It consists of using a set of equations considered as left-to-right rewrite rules to simplify a given term, called the *subject term*. Starting from this term, the evaluation process produces a sequence of expressions by repeatedly replacing instances of left sides of rules with their corresponding right sides until no further replacements are possible. An instance of a left side in the subject expression is called a *redex*, and an expression with no redex is said to be in *normal form*.

The pattern-matching process provides a rule whose left side matches the expression considered. As patterns can overlap, several rules can be matched at the same time. In this case, a meta-rule allowing for the selection of a single matched rule is used. Examples of such a meta-rule are the first matched rule (textual order) and the most specific matched rule [4]. The textual order meta-rule will be used here in the case of overlapping patterns. However,

[*] Correspondence to: N. Nedjah, Computation Department, UMIST, PO Box 88, Manchester M60 lQD, UK.

[¶] The author was initially sponsored by the British Council and the Algerian High Education Ministry, and is presently sponsored by the Fundação de Amparo à Pesquisa no Estado do Rio de Janeiro (FAPERJ). On leave of absence from Institut d'Informatique, Université de Annaba, B.P. 12, Annaba 23000, AIgerie.

CCC 0038–0644/99/090793–21$17.50

Received 1 *September* 1996
Revised 23 *December* 1998
Accepted 17 *February* 1999

the method presented through this paper is easily adapted to any meta-rule which can be established at compile-time.

Usually, the pattern set is pre-processed producing an intermediate representation allowing for the matching process to be performed efficiently. Different kinds of such representations have been studied: conditional constructs, like those in procedural languages, have been exploited for this purpose. Examples include the *if-then-else* construct [5] and the *case-expression* [6,7]. Another kind of representation consists of a *matching* tree (also called the index tree) [3,8]. Similarly, pattern-matching definitions have been compiled into a finite matching automaton in References [1,9,10,11,12].

Pattern-matching automata have been studied for over a decade. Gräf [9], Maranget [13] and Christian [14] describe matching automata for unambiguous patterns based on left-to-right traversal. Gräf [9] adds instances of patterns using a closure operation, so symbol re-examination could be avoided. Maranget [13] describes two techniques to compile lazy pattern-matching. The first technique generates a deterministic matching automaton that is equivalent to that obtained by Gräf [9]. The second technique, however, generates automata with *failures* that allow non-deterministic pattern-matching (i.e. with symbol re-examination). These automata possess a static exception construct that enables some code sharing. In functional programming, Augustsson [6] and Wadler [7] describe pattern-matching techniques that are also based on left-to-right traversal, but allow prioritised overlapping patterns. They compile patterns into *CASE* constructs using four compilation rules: the *empty rule*, the *constructor rule*, the *variable rule* and the *mixture rule*. Although the latter methods are practical and economical in terms of space usage, they may re-examine symbols in the input term. In the worst case, these methods can degenerate to the naive method of checking the input term against each pattern individually. In contrast, Christian's [14] and Gräf's [9] methods, together with Maranget's [13] first technique, avoid symbol re-examination at the cost of increasing the space requirements, while Maranget's [13] second technique guarantee that the size the resulting automaton is linear in the size of the patterns. However, one has to bear in mind that the functional approach followed by Wadler [7] and Maranget [13] combines the operations of forcing weak normalisation of the subterm whose the head symbol mismatches the symbol in the pattern and examining the resulting head symbol while in the term rewriting approach followed by Christian [14] and Gräf [9], symbols are examined without forcing any evaluation.

The practical method we shall describe in this paper is similar to a method of automatic generation of parsers [15,16], which has been used for many years to compile imperative languages. Backtracking in the pattern-matching process can consume considerable time as well as space. So, through the matching table generation process, we convert the original pattern set to an equivalent closed pattern set which avoids the need for backtracking. We use a different approach from that described by Gräf [9] to compute the closure of pattern sets. In contrast with his method, we also deal with prioritised overlapping patterns. Furthermore, Gräf [9] does not show how to use the constructed matching automaton in the context of a given rewriting strategy. After the method of generating the matching table is described, we describe how to interpret such a matching table through an abstract rewriting machine. We show that the rewriting machine is simple yet expressive, and any rewriting strategy can be used after minor changes to it. We illustrate this machine using the three most popular strategies namely, leftmost-innermost [17,18], leftmost-outermost [12] and the adaptive strategy [19] used in most lazy functional languages such as Miranda [19], LML [20] and Haskell [21]. The adaptive strategy implements the functional approach described in Reference [7]. Our method, like most others, is restricted to the subclass of left-linear programs for which a variable can occur only once in the left side of an equation.

Using the technique of partial evaluation [22], Strandh [3] transforms the code generated for equational programs so that the code obtained allows for some matching and rewriting steps to be avoided. Similar work can be found in References [23-25]. Instead, we customise the idea of partial evaluation so it can be used directly on the equational program itself, not on the code generated for it. For the adaptive strategy, it is possible to avoid some matching and rewriting steps if the equation's right sides are analysed at compile-time. The rewriting machine permits this analysis to be done easily, and it generates a new equivalent set of equations which is more efficient. Finally, an evaluation of an implementation of the proposed ideas is provided.

2. NOTATION AND DEFINITIONS

Definition 2.1. An *equational program* can be defined as a 4-tuple $EP = \langle F, V, R, T \rangle$ where $F = \{f, g, h, ...\}$ is a set of function symbols, $V = \{x, y, z, ...\}$ is a set of variable symbols and $R = \{ \pi_1 \rightarrow \tau_1, \pi_2 \rightarrow \tau_2, ...\}$ is a set of rewrite rules called the *term rewriting system*, where π_i and τ_i are terms, called the *pattern* and *template*, respectively. T is the *subject term*, which is the expression to evaluate.

For convenience, in most of the paper, we will consider the patterns to be written from the symbols in $F \cup \{\omega\}$, where ω is a meta-symbol used whenever the symbol representing a variable does not matter. For a pattern set π, we denote by F_π the subset of F containing only the function symbols in the patterns of π. A *term* is either a variable, a constant symbol or has the form $f(\sigma_1, \sigma_2, ..., \sigma_n)$ where each σ_i $(1 \leq i \leq n)$ is itself a term and n is the arity of the function symbol f denoted by $\#f$. The subject term is supposed to be a *ground* term, i.e. a term containing no variable occurrences. Terms are interpreted syntactically as trees labelled with symbols from $F \cup V$. An *instance* of a term t can be obtained by replacing leaves labelled with variable symbols by other terms. In practice, however, both the subject term and templates are turned into Directed Acyclic Graphs (DAGs) so that common subterms may be shared (represented physically once). This allows for the evaluation of such subterms to be performed at most once during the whole rewriting process.

Definition 2.2. A *position* in a term is a path specification which identifies a node in the graph of that term, and therefore both the subterm rooted at that point and the symbol which labels that node. A position is specified here using a list of positive integers. The empty list Λ denotes the graph root, the position k denotes the kth child of the root, and the position $p.k$ denotes the kth ($k \geq 1$) child from the position given by p. The symbol, respectively subterm, rooted at position p in a term t is denoted by $t[p]$, respectively t / p. A position in a term is *valid* if, and only if, the term has a symbol at that position. So Λ is valid for any term, and a position $p = q.k$ is valid if, and only if, the position q is valid, the symbol f at q is a function symbol and $k \leq \#f$.

For instance, in the term $t = f(g(a, h(a, a), b(a, x), c)$, $t[\Lambda]$ denotes the single occurrence of f, $t[2.2]$ denotes the variable symbol x, whereas $t[2]$ denotes the symbol b while $t/2$ indicates the subterm $b(a,x)$ and the positions 2.2.1 and 1.3 are not valid. In the following, we will abbreviate terms by removing parentheses and commas. For instance, t abbreviates to $fgahaabaac$. This will be unambiguous, since the given function arities (i.e. $\#f = 3$, $\#g = \#h = \#b = 2$, $\#a = \#c = 0$) will be kept unchanged throughout all examples. In particular, the arities $\#f = 3$, $\#g = 2$ and $\#a = 0$ will be used in the running example.

Definition 2.3. A pattern set π is overlapping if there is a ground term that is an instance of at least two distinct patterns in π.

For instance, the set $\pi = \{fa\omega\omega, f\omega a\omega\}$ is an overlapping pattern set because the term *faac* is an instance of both patterns, whereas the set $\pi' = \{fa\omega\omega, fc\omega\omega\}$ is a non-overlapping pattern set. A similar notion is that of pattern prefix-overlapping:

Definition 2.4. A pattern set π is *prefix-overlapping* if there is a ground term with a non-empty prefix that is an instance of prefixes of at least two distinct patterns in π.

For instance, the set $\pi = \{f\omega aa, f\omega\omega c\}$ is a non-overlapping pattern set, but it is a prefix-overlapping because the prefix *faa* of the term *faaa* is an instance of both prefixes $f\omega a$ and $f\omega\omega$.

When overlapping patterns are allowed in equational programming, a meta-rule is needed to decide which rule should be matched when a conflict due to overlapping patterns arises. The meta-rule defines a priority relationship among overlapping patterns. Thus, given a pattern set π and a meta-rule, we can formalise the notion of pattern-matching as follows:

Definition 2.5. A term t *matches* a pattern $\pi_i \in \pi$ if and only if t is an instance of π_i and t is not an instance of any pattern $\pi_j \in \pi$ such that the priority of π_j is higher than that of π_i .

Definition 2.6. A term t_1 is *more general* than a term t_2 at a given common valid position p if and only if $t_1[p] \in V$, $t_2[p] \in F$ and the prefixes of t_1 and t_2 ending immediately before P are the same.

Definition 2.7. The *closed pattern set* $\overline{\pi}$ corresponding to a given pattern set π is the set obtained by applying to π the closure operation defined by Gräf [9] as follows:

For any $s \in F \cup \{\omega\}$, let π / s be the set of elements of π starting with s but with the first symbol s removed. Define π_ω and π_f, $f \in F$ by

$$\pi_\omega = \pi / \omega$$

$$\pi_f = \begin{cases} \pi / f \cup \omega^{\#f}\pi / \omega & \text{if } \pi / f \neq \emptyset \\ \emptyset & \text{otherwise.} \end{cases}$$

The closure operation is then defined recursively by:

$$\overline{\pi} = \begin{cases} \pi & \text{if } \pi = \{\varepsilon\} \text{ or } \pi = \emptyset \\ \bigcup_{s \in F \cup \{\omega\}} s\overline{\pi_s} & \text{otherwise.} \end{cases}$$

Here ε is the empty string and $\omega^{\#f}$ is a repetition of $\#f$ symbols ω. The set π_f includes all the patterns in π starting with f, but with f removed. In addition, while *factorising* a pattern set according to a function symbol f (i.e. computing π_f), the operation above takes account of the components starting with a variable symbol as well; the symbol ω is considered as possibly representing a subterm whose root symbol is f. Therefore, a new component is added to the set π_f. This component is obtained by replacing ω by a sequence of ωs whose length is $\#f$. This sequence stands for the arguments of f.

The closure operation supplies the pattern set with some instances of the original patterns. In effect, if one pattern is more general than another at some position p then the pattern with ω replaced by $f\omega^{\#f}$ is added. For instance, the prefix-overlapping pattern set $\pi = \{f\omega a\omega, f\omega\omega a, fg\omega\omega g\omega\omega\}$ can be converted to an equivalent closed pattern set π using the closure operation computed as follows:

$$\overline{\pi} = f\overline{\pi_f} = f\{\overline{\omega a\omega, \omega\omega a, \omega g\omega\omega g\omega\omega}\}$$
$$= f\omega\{\overline{a\omega, \omega a, g\omega\omega g\omega\omega}\} = f\omega\overline{P},$$

where

$$\begin{aligned}\overline{P} &= a\overline{P_a} \cup \omega\overline{P_\omega} \cup g\overline{P_g} \\ &= a\{\overline{\omega, a}\} \cup \omega\{\overline{a}\} \cup g\{\overline{\omega\omega g\omega\omega, \omega\omega a}\} \\ &= \{a\omega, aa, \omega a, g\omega\omega g\omega\omega, g\omega\omega a\}.\end{aligned}$$

Then, the closed pattern set corresponding to π is:

$$\overline{\pi} = \{f\omega aa, f\omega a\omega, f\omega g\omega\omega a, f\omega g\omega\omega g\omega\omega, f\omega\omega a\}.$$

It is clear that the new pattern set accepts the same language as π does, since the added patterns are all instances of the original ones. The closure operation terminates and can be computed incrementally (for full detailed description and formal proofs see Reference [9]).

With closed pattern sets, if a pattern π_1 is more general than a pattern π_2 at position p, then $\pi_2[p]$ is checked first. This does not exclude a match for π_1 because the closed pattern set does contain a pattern that is π_1 with $\pi_1[p]$ replaced by $\pi_2[p]\omega^{\#\pi_2[p]}$. Under this assumption, an important property of such closed pattern sets is that they make it possible to determine whether a target term is a redex merely by scanning that term from left-to-right without backtracking over its symbols [9].

Symbol re-examination cannot be avoided in the case of non-closed prefix-overlapping pattern sets whatever the order in which the patterns are provided. For instance, let π be the prefix-overlapping set $\{fc\omega c, f\omega ga\omega a\}$. Using the textual order meta-rule, the first pattern must be matched first, if possible. Then the term *fcgaaa* cannot be identified as an instance of the second pattern without backtracking to the first occurrence of *c* when the last symbol *a* is encountered. However, the closure $\overline{\pi} = \{fcga\omega c, fcga\omega a, fc g\omega\omega c, fc\omega c, f\omega ga\omega a\}$ allows matching without backtracking. Then the term will match the second pattern.

3. MATCHING TABLE GENERATION

In this section, we describe a different approach to compute the closure of a pattern set π via direct compilation rather than via the sets π_f, as in Reference [9] (see Definition 2.7). We compile pattern sets into matching tables. In general, the pattern-matching compilation technique at the heart of this paper can be thought of as a table-driven method inspired by the LALR method used in YACC [15,16] to generate parsers for LR-languages. In that context, the pattern set to be compiled is considered as the set of right sides of syntactic productions. The left sides of these productions are non-terminals representing the *types* of the patterns, and each variable symbol is a non-terminal for its *type*. Since we are dealing only with untyped systems, there is only one non-terminal. As with YACC [15,16] the compilation process creates a finite automaton, for which we must define the states and the state transition function.

Definition 3.1. A *matching item* is a pattern which is split into a prefix and suffix by inserting the *matching dot* ($\cdot$) at some point.

In general, for the pattern $\alpha\beta$ the matching item $\alpha \cdot \beta$ means that the symbols in the prefix α have been matched and those in β have not been checked yet. Thus, the matching item $\cdot\beta$ represents the initial state prior to matching the pattern β, whilst the matching item $\alpha\cdot$ represents the final state after matching the whole pattern α.

Definition 3.2. A set of matching items in which every item has the same prefix before the matching dot is called a *matching set*. The *initial matching set* contains all the matching items

$\cdot\pi_i$ s.t. $\pi_i \in \pi$, whereas matching sets containing items of the form $\pi_i\cdot$ are *final matching sets*. Note that final matching sets can contain only one matching item.

A pattern set π is compiled into a deterministic finite matching automaton. The states of this matching automaton are computed using the following transition operation δ. For each symbol $s \in F \cup \{\omega\}$ and matching set I, a new matching set $\delta(I, s)$ is defined by

$$\delta(I,s) = \{\alpha s\cdot\beta \mid \alpha\cdot s\beta \in I\} \cup \{\alpha s\cdot f\omega^{\#f}\mu \mid \alpha\cdot s\omega\mu \in I \text{ and for some } f \in F \text{ and term } \mu',\ \alpha\cdot sf\mu' \in I\}$$

Notice that the presence of the items $\alpha s\cdot\omega\mu$ together with the items $\alpha s\cdot f\mu'$ in the same matching set creates a non-deterministic situation for a pattern-matcher, since ω can be substituted with a term having f as head symbol. The items $\alpha s\cdot f\omega^{\#f}\mu$ are added to remove such non-determinism and avoid backtracking. For instance, let $\pi = \{f\omega\omega a, fc\omega c\}$, and let M be the matching set obtained after accepting the root symbol f so $M = \{f\cdot\omega\omega a, f\cdot c\omega c\} \cup \{f\cdot c\omega a\}$. The item $f\cdot c\omega a$ is added because a target term with the prefix fc could match the pattern $f\omega\omega a$ too if the last argument of f were a rather than c. So supplying the instance $fc\omega a$ would allow the pattern-matcher to decide deterministically which option to take. Without this new item, the pattern-matcher would need to backtrack to the first argument of f if the option offered by $f\omega\omega a$ were taken, and a symbol c encountered as the last argument of f in the target term. Notice that the transition operation thus described implements exactly the main step in the closure operation due to Gräf [9], but replaces his recursive description with a straightforward iterative construction. Therefore, the union of the final pattern sets resulting from the automaton construction procedure coincides with the closure of the initial pattern set.

Each matching set is associated with a state in the matching automaton, namely that accepting the common prefix (before the matching dot) in the given pattern set. The initial state and final states correspond to the initial matching set and final matching sets, respectively. The edges of the finite automaton are traversed according to the current input symbol, namely s causes the transition from I to $\delta(I, s)$. The matching automaton corresponding to $\pi = \{f\omega a\omega, f\omega\omega a, f\omega g\omega\omega g\omega\omega\}$ is given in Figure 1. Transitions corresponding to failure are omitted.

The finite matching automaton corresponding to a given pattern set is represented using a matching table. Matching tables are simple, compact and expressive. Also, they allow a direct access to a given matching state (see Section 5). A matching table is an $N \times (L + 1)$ matrix of transitions, where N is the number of non-final states in the automaton (10 in the example of Figure 1) and L is the number of function symbols in F_π. The extra column of N entries is used for variable occurrences in the patterns, all of which are denoted by ω. The distinction between ω and the other symbols permits a concise representation of the matching table. This enables the use of matching tables for equational programs with an infinite alphabet.

For a matching state I and a symbol $s \in F_\pi \cup \{\omega\}$, let $\delta(I, s) = J$. Then the matching table entry $MT[I, s]$ is

accept-symbol$_J$	if J is not a final state and $s \in F_\pi$.
accept-term$_J$	if J is not a final state and $s = \omega$.
reduce$_r$	if J is a final state and r is the matched rule (see the next section).
fail	Otherwise, i.e. J is empty.

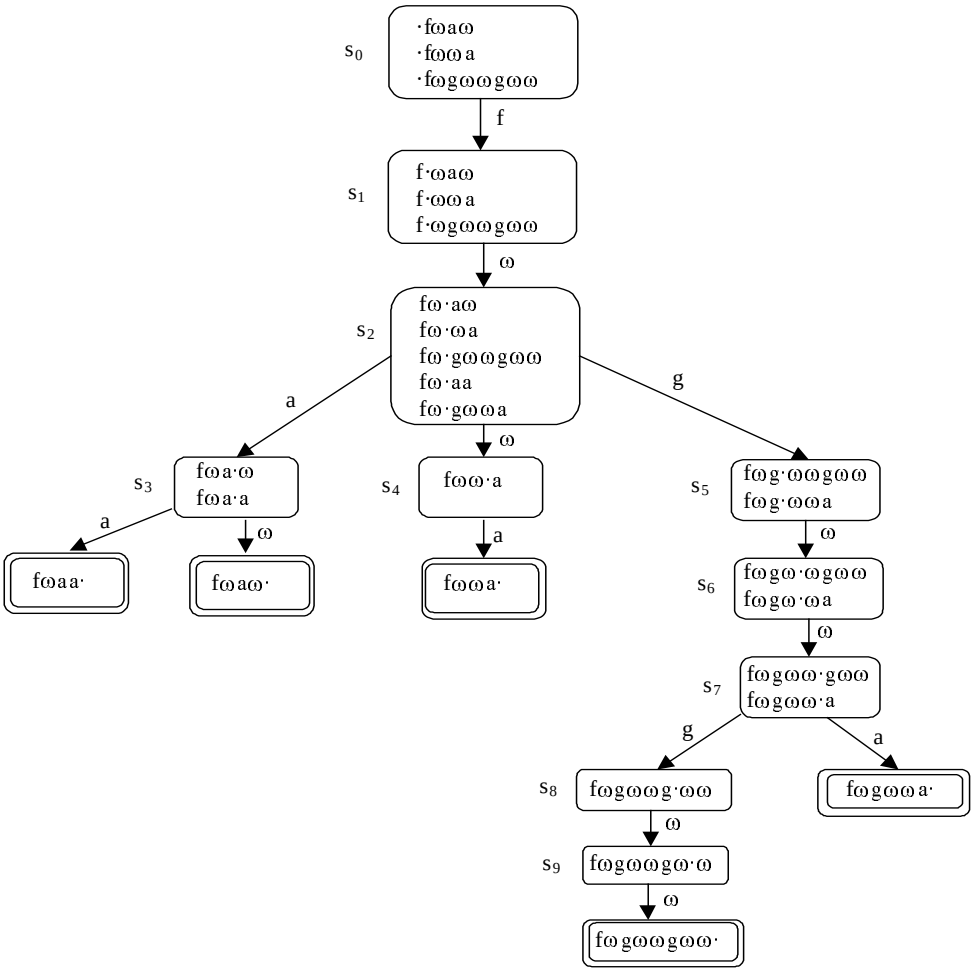

Figure 1. *Pattern-matching automaton for* $\pi = \{f\omega a\omega, f\omega\omega a, f\omega g\omega\omega g\omega\omega\}$

Here the subscript *J* indicates the next state to enter. *Accept-symbol* and *accept-term* mean that the current input symbol (respectively the current input term) is matched, and the state $\delta(I, s)$ is entered. *Reduce*$_r$ means that the rule number *r* has been matched and should be applied, whereas *fail* means matching has failed at the given symbol *s*.

The rules given are used straightforwardly. In all cases, an ω-transition is chosen only when the matching table entry for the specific current input symbol is *fail*. So if $s \in F_\pi$ is the current input symbol and *I* the current state, then action $MT[I, f]$ is performed if it is not a *fail*, and otherwise $MT[I, \omega]$ is performed. Setting an entry to *reduce*$_r$ requires solving an additional problem if the matched pattern has been added by the construction process. For original patterns the rule number to use is uniquely defined. However, two or more original patterns could match any added patterns. The selection is made by applying the rule priorities provided to associate a unique rule with each added pattern.

For instance, for the pattern set $\pi = \{1: f\omega a\omega, 2: f\omega\omega a, 3: f\omega g\omega\omega g\omega\omega\}$ of Figure 1, the added patterns $f\omega aa$ and $f\omega g\omega\omega a$ would be associated respectively with the rule numbers 1 and 2 if the textual order rule were used.

Table I. Matching table for $\pi = \{f\omega a\omega, f\omega\omega a, f\omega g\omega\omega g\omega\omega\}$

	f	*a*	*g*	ω
0	*accept-symbol*$_1$	*fail*	*fail*	*fail*
1	*fail*	"	"	*accept-term*$_2$
2	"	*accept-symbol*$_3$	*accept-symbol*$_5$	*accept-term*$_4$
3	"	*reduce*$_1$	*fail*	*reduce*$_1$
4	"	*reduce*$_2$	"	*fail*
5	"	*fail*	"	*accept-term*$_6$
6	"	"	"	*accept-term*$_7$
7	"	*reduce*$_2$	*accept-symbol*$_8$	*fail*
8	"	*fail*	*fail*	*accept-term*$_9$
9	"	"	"	*reduce*$_3$

Ambiguity arises for patterns introduced that belong to the overlap of any original patterns. For example, let π_3 be the pattern added because pattern π_1 is more general than pattern π_2 at a given position p. Recall that π_3 is obtained from π_1 by replacing the subterm $\pi_1[p] = \omega$ by a term of which the subterm $\pi_2[p]$ is an instance. Thus, π_3 is an instance of π_1. Normally, π_3 will fail to overlap with π_2, and so the rule for π_1 will also be associated with π_3. However, if π_1 and π_2 overlap, π_3 may also be an instance of π_2. Therefore, the template associated with either π_1 or π_2 could be selected to rewrite the subexpression being evaluated. The use of the given meta-rule allows for the selection of such a template. This is performed at compile-time by associating the prescribed rule number among those corresponding to π_1 and π_2, to the added pattern π_3.

The matching table corresponding to the pattern set π of Figure 1 is shown in Table I. It corresponds to the finite automaton of that figure. For convenience, the states/matching sets have been numbered as before. Notice that rows representing final states do not exist in the matching table. The decision to reduce the target expression using the matched rewriting rule is anticipated in the state leading to the corresponding final state. This allows for the reduction of the matching table by r rows, where the number r of final states is also the number of patterns in the closure of the original pattern set.

4. REWRITING MACHINE APPLICATION

In this section, we show how the matching table constructed so far can be used in rewriting. This detail was not covered by Gräf [9] for his associated finite automaton. We illustrate the rewriting process using the three most popular reduction strategies, namely leftmostinnermost, which is used for strict functional languages such as HOPE [17], and ML [18] leftmost-outermost [12], and the *adaptive* strategy which is used in most lazy functional languages such as LML [20], Haskell [21] and Miranda [19]. The last strategy is sometimes called 'top-to-bottom left-to-right lazy strategy' [4], and is defined after the next example.

Consider the prioritised equational program, which has the following set of rewrite rules with the subject term $t = f(c, f(a, a, a), a)$:

$$\begin{aligned} f(x, a, y) &\rightarrow y \quad (r_1) \\ f(x, y, a) &\rightarrow a \quad (r_2) \\ c &\rightarrow c \quad (r_3) \end{aligned}$$

Closure would add the pattern $f(x, a, a)$ which overlaps both left sides. Then assuming a textual order meta-rule, $f(x, a, a)$ would be associated with r_1. The leftmost-outermost

strategy would reduce t using r_2, and a would be the normal form obtained whilst the leftmost-innermost (or *applicative*) strategy would result in an infinite computation, since it would cause the repeated reduction of c. Arguably, the reduction sequence that best captures the semantics [19] of the prioritised system above is as follows. Contracted redexes are underlined:

$$f(c, \underline{f(a, a, a)}, a) \xrightarrow{r_2} \underline{f(c, a, a)} \xrightarrow{r_1} a$$

This strategy, which we will call the *adaptive* strategy, selects the leftmost-outermost redex after reducing the arguments needed to normal form. That is, if during pattern-matching the root symbol of a subterm fails to match a function symbol f in the pattern, then the subterm is evaluated before re-attempting the match. If the root symbol of the normal form of the subterm is different from f, the matching of that pattern fails and the next pattern is tried.

In some cases, the adaptive strategy terminates when the leftmost-outermost strategy does not. For instance, consider the rewriting system $\{f(x, a, y) \rightarrow a, g(x, y) \rightarrow a, c \rightarrow c\}$. The evaluation of the term $f(c, g(a, a), c)$ using the leftmost-outermost strategy fails to terminate, since it would try to rewrite the subterm c that represents the outermost-leftmost redex. However, the adaptive strategy would try to match the pattern of the first rule by rewriting the second argument of f to a. It does this, and hence succeeds in rewriting $f(c, g(a, a), c)$ to a. The adaptive strategy requires equational programs to follow the *constructor discipline* [26]. This can be formalised as follows:

Definition 4.1. A function symbol is a *constructor* for the pattern set π if it does not appear at the root of any pattern in π.

Definition 4.2. Let R be a term rewriting system, and suppose every symbol position in a pattern of R, other than the first, is a constructor symbol for the set of patterns of R. Then R is called a *constructor* system.

For instance, consider the term rewriting system (E) that follows. The initial symbols f, g and h are not constructors, whereas the symbols a and b are. Therefore, (E) is a constructor system.

1. $f(x, a, y) = a$
2. $f(x, y, a) = a$
3. $f(x, b(s, y), b(t, z)) = h(b(y, a), g(x, a))$
4. $g(x, a) = x$
5. $h(b(x, a), a) = a$
6. $h(x, a) = g(a, x)$

The constructor discipline entails almost no loss of generality, since the majority of equational programs observes this discipline [12] and the majority of those that do not, can be syntactically transformed so as to follow it [26,27]. For the remainder of this paper, we assume, in common with most researchers in the area, that the term rewriting system is a constructor system.

The pattern-matching process is incorporated into a simple abstract rewriting machine represented by the 5-tuple $RM = \langle I, P, MT, R, T \rangle$, which will be used to rewrite the term T to its normal form. I is the current matching state which corresponds to a row index in the matching table MT. R represents the root position of the subterm in T currently being matched. P is the current position the matching process has reached in the subterm $T[R]$.

Now, let $\langle I, P, MT, R, T \rangle$ be the current state of RM. Then its next state is determined according to the reduction strategy used, $MT[I, T[R.P]]$ and $MT[I, \omega]$. Possible state transition rules for the abstract machine using the leftmost-innermost and leftmost-outermost strategies are described in Figures 2 and 3 respectively. The transition rule of Figure 2 uses the adaptive strategy. In each of these three figures, *MatchingAction* represents the matching

action to be performed. This is either $MT[I, T[R.P]]$ or *fail*, depending on whether or not the symbol $T[R.P]$ is a function symbol that actually appears in a pattern.

```
⟨I, P, MT, R, T⟩ →
    Case MatchingAction of
        accept-symbol_J: If U = {i | 1 ≤ i ≤ #T[R.P], ¬IsNormal(P.i) } ≠ ∅ then
                         For i ∈ U do Normalise(T/R.P.i)
                         else ⟨J, Next(P), MT, R, t⟩;
        reduce_r:        ⟨0, Λ, MT, Λ, ApplyRule_r(T, R)⟩;
        fail:            Case MT[I,ω] of
                             accept-term_J:  ⟨J, NextArg(P), MT, R, T⟩;
                             reduce_r:       ⟨0, Λ, MT, Λ, ApplyRule_r(T, R)⟩;
                             fail:           If Next(R) Defined then ⟨0, Λ, MT, Next(R), T⟩
                                             else MarkAsNormal(T);
                         End
    End
    Where MatchingAction = If T[R.P] ∈ F_π then MT[I, T[R.P]] else fail
```

Figure 2. *State transition rule for innermost-leftmost strategy*

All of these strategies require each subterm of T to be marked initially as not normal (i.e. not known to be in normal form). The normal form of a given term T is obtained by applying the appropriate state transition rule repeatedly until the root of T is marked as normal. Such an iteration sequence will be called *Normalise*(T). It starts in the initial state $\langle 0, \Lambda, MT, \Lambda, T\rangle$, with all the nodes in the graph of T marked as not normal.

```
⟨I, P, MT, R, T⟩ →
    Case MatchingAction of
        accept-symbol_J: ⟨J, Next(P), MT, R, T⟩;
        reduce_r:        ⟨0, Λ, MT , Λ, ApplyRule_r(T, R)⟩;
        fail:            Case MT[I,ω] of
                             accept-term_J:  ⟨J, NextArg(P), MT, R, T⟩;
                             reduce_r:       ⟨0, Λ, MT, Λ, ApplyRule_r(T, R)⟩;
                             fail:           If Next(R) Defined then ⟨0, Λ, MT, Next(R), T⟩
                                             else MarkAsNormal(T);
                         End
    End
    Where MatchingAction = If T[R.P] ∈ F_π then MT[I, T[R.P]] else fail
```

Figure 3. *State transition rule for leftmost-outermost strategy*

The function *Next*(P) returns the position of the symbol after that at position P in the term T whereas *NextArg*(P) returns the position of the next argument (i.e. the next sibling in that

graph). Here *NextArg* will travel back up towards the root of *T* to find the lowest ancestor of *P* which has a branch to the right of *P* .

For instance, let *T* = *fagaahaa*. Then applying the functions *Next* to positions Λ, 2, 2.1 and 2.2 returns the positions 1, 2.1, 2.2 and 3, respectively. For both of positions 2 and 2.2, function *NextArg* returns position 3. The function *Next* is undefined for the position of the last symbol in the term, whilst *NextArg* is undefined for all positions between the root and the right-most leaf of the parse tree of *T*. Regarding the example term *T* above, *Next* is undefined for position 3.2 while *NextArg* is undefined for positions 3.2 and 3. Where *NextArg* is used in the transition rules, it is always defined, but when *Next*(*R*) becomes undefined the machine halts.

In the transition rule for the adaptive strategy, when a rule is applied, *R* remains the root of the subterm in which the next redex is searched for, whereas for the other two strategies, the new root *R* becomes Λ. *Apply-rule*$_r$(*T*, *R*) instantiates the template of the matched rule *r*, replaces the variable nodes by their actual graph-values, and finally, rewrites the subterm *T* / *R* using the newly constructed template instance. Given a position *P*, the Boolean function *IsNormal* checks whether the graph rooted at position *P* is known to be in normal form, whereas the function *MarkAsNormal*(*T*) marks the root of the term *T* when this term is known to be in normal form. Eventually, the machine halts at (0, Λ, *MT*, *L*, *T*), where *T* is marked as normal and *L* is the position of the last symbol in the term *T*.

The machine state transition rules of Figures 2 and 3 are used in the obvious way. For the adaptive strategy, whenever a function symbol *f* labels the current matching position *P* and the entry *MT*[*I*, *f*] is *fail*, two alternatives are possible according to whether there is another function symbol *g* for which *MT*[*I*, *g*] is not *fail*. When such a symbol exists, the subterm rooted at position *P* is *normalised* then, the transition rule tried again (in the hope that *g* might be matched). Otherwise, the entry *MT* [*I*, ω] is tried.

For instance, consider a rewriting system for which the matching table is that of Table I. Suppose rule 1 is *fxay* → *a* and the subject term is *T* = *fcfcaac*. Using the transition rule of Figure 2 first causes the normalisation of argument *fcaa* of the root symbol *f*, since the entry *MT*[0, *f*] is *accept-symbol*$_1$ and not all of the arguments of the first *f* in *T* are in normal form. The subterm *fcaa* readily matches the pattern of rule 1 so *T* reduces to *fcac*. Then state 1 is entered. Once the prefix *fca* is accepted, *I* and *P* become 3 and 3, respectively. Subsequently, with *c* at *T*[3], *MT*[3, ω] indicates that the pattern of rule 1 is matched, so *T* reduces to *a*. The same outcome arises from the use of the transition rule of Figure 3. There the prefix *fc* is accepted and state 2 entered; then the subterm *fcaa* is skipped because *MT*[2, *f*] is fail and so *MT*[2, ω] is performed. Thus state 4 is entered, and so matching at the root fails. Subsequently, the strategy fails to find a redex at position 1, but succeeds in identifying the redex *fcaa* at position 2. This readily matches the pattern of rule 1, and is rewritten to *a*, so *T* reduces to *fcac*. Once again, the strategy succeeds in matching the pattern of rule 1 at position Λ so that *T* reduces to *a*. A similar result obtains from using the transition rule of Figure 4. This accepts the prefix *fc*; enters state 2 with *P* = 2; since *MT*[2, *f*] is fail and *T*[*P*] is not in normal form, the normalisation of subterm *fcaa* takes place so that *T* reduces to *fcac*; the match of the symbol at position 2 is re-attempted; the symbol *T*[2] = *a* is accepted and state 3 entered; with the normal form *c* at position 3, *MT*[3, ω] indicates that the pattern of rule 1 is matched. The adaptive strategy then applies that rule, and so *T* reduces to *a*.

When a pattern is matched the pattern-matching process needs to provide the actual substitutions for ωs in that pattern. This is to enable template instantiation so that redexes can be reduced. Gräf's technique [9] does not mention such a detail. Since variable occurrences in patterns are known, their positions can be pre-computed and stored at compile-time. Once a rule has been matched and the value of a variable is needed, the subterm corresponding to the

position of that variable can be retrieved from the target expression. This is then provided to the template instantiating process which uses it to replace occurrences of the variable.

$\langle$I, P, MT, R, T$\rangle$ $\rightarrow$
  Case *MatchingAction* **of**
    accept-symbol$_J$: $\langle$J, *Next*(P), MT, R, T$\rangle$;
    reduce$_r$: $\langle 0, \Lambda, MT, R, ApplyRule_r(T, R)\rangle$;
    fail: **If** T[R.P]$\in F_\pi$ **and** $\exists f \in F$ (MT[I,*f*]$\neq$fail) **and** $\neg$*IsNormal*(P) **then**
      Normalise(T/R.P)
    else Case MT[I,ω] **of**
      accept-term$_J$: $\langle$J, *NextArg*(P), MT, R, T$\rangle$;
      reduce$_r$: $\langle 0, \Lambda, MT, R, ApplyRule_r(T, R)\rangle$;
      fail: **If** *Next*(*R*) Defined **then** $\langle 0, \Lambda, MT,$ *Next*(R), T$\rangle$
        else *MarkAsNormal*(T);
    End
  End
  Where *MatchingAction* = **If** T[R.P] $\in F_\pi$ **then** MT[I, T[R.P]] **else** fail

Figure 4. *Machine state transition rule for the adaptive strategy*

However, as backtracking has been eliminated, the cost of dynamically collecting pointers to variable instances during pattern-matching may be cheaper. Let $\langle I, P, MT, R, T\rangle$ be the current state of *RM*. It is clear that when the matching action to perform is $MT[I, \omega]$ (i.e. either *accept-term* or *reduce*), the subterm $T / R.P$ is the actual value of the variable $\pi[P]$ if matching of pattern π succeeds. This collects substitutions for the matched pattern, not for the original pattern in the rule which is to be applied. For instance, consider the pattern set of Figure 1 and let *faaa* be the subject term. Because *faaa* would match the non-original pattern $f\omega aa$, only one variable substitution would be available when matching concludes. However, *faaa* has to be rewritten using rule 1 whose pattern $f\omega a\omega$ requires two substitutions, not one. In this case, the missing substitution is always the instance *a* at position 3 in the matched pattern.

To be able to collect the missing substitutions when non-original patterns are matched, we need to consider some cases for which the matching action to perform is $MT[I, f]$ for $f \in F_\pi$. When the matching action to perform is $MT[I, f]$, a substitution needs to be collected only if $MT[I, \omega]$ is distinct from *fail*. Then in the example above, two additional variable instances would be collected when the second and last symbols *a* in *faaa* are accepted in states 2 and 3, respectively. This is because the actions to be performed are $MT[2, a] = accept\text{-}symbol_3$ and $MT[3, a] = reduce_1$ with $MT[2, \omega] = accept\text{-}term_4$ and $MT[3, \omega] = reduce_1$, respectively, but not *fail*. No substitution would be collected when the symbol *f* is accepted in state 0, because $MT[0, \omega] = fail$. The substitution for the second occurrence of *a* is collected because *faaa* is an instance of the pattern $f\omega\omega a$ (rule 2) too. Then the required substitutions would be available if rule 2 were to be applied (in case rule 2 has higher priority than rule 1).

In some cases, even if the target term can match only one original pattern, there may be more variable substitutions than are required. For instance, when the non-original pattern $f\omega g\omega\omega a$ in Figure 1 is matched, four variable instances would be collected, namely those

rooted at positions 1, 2, 2.1 and 2.2 in the target term: three correspond to the occurrences of ω and one to the occurrence of g. No substitutions are collected for the occurrences of f and a in states 0 and 4, respectively, because $MT[0, \omega]$ and $MT[4, \omega]$ are *fail*. Since $f\omega g\omega\omega a$ is associated with rule 2 whose pattern is $f\omega\omega a$, only the variable instances rooted at positions 1 and 2 would be used, and the rest ignored.

5. ANALYSIS OF RIGHT-HAND SIDES

Exploiting the idea of partial evaluation in the compilation of equational programs, we can feed the templates into a partial evaluator for further processing. The result may enable the skipping of some matching steps every time a rewrite rule is performed. In the best case, the construction of some terms will be avoided. In this section, we describe how to take advantage of known information in the templates by partially evaluating them at compile-time.

In general, partial evaluation consists of transforming a given program *Prog* (perhaps using some partial input) into another program *Prog'* which produces the same result. The main transformations [22] are known as *constant folding*, *function specialisation* and *call unfolding*. Constant folding simplifies expressions by replacing known subexpressions by their values, while function specialisation specialises function definitions to take advantage of some information concerning some of its arguments. Finally, call unfolding unfolds function calls to expose it to some improvements due to a particular calling context.

Partial evaluation for equational programming was first introduced by Strandh [3] and continued by Durand [23,24], Sherman [25] and Miniussi [28]. In most of these works, the equational program is compiled into intermediate code, then using the information known about the equations' right sides, this code is specialised by means of the transformations above. The specialised code avoids the construction of some nodes and the checking of a known part of the templates. However, rather than applying the transformations to the code generated for the equational program, we instead apply them instead directly to the rewrite rules of the program.

The right-hand side analysis at compile-time exploits the first two transformations, namely *constant folding* and *function specialisation*, (i) to transform the templates themselves so the construction of some terms will be avoided at run-time, and (ii) to specialise the pattern--matcher so that it avoids checking the known part of the template. The analysis consists of matching and rewriting the templates as far as they allow it using the matching automaton. This process halts when the template considered is not defined enough to continue matching or rewriting. Then the current matching state K and position Q are returned with the current template. This new template decorated with K and Q is used to replace the appropriate rewrite rule right side in the original equation system. K and Q are used to specialise the pattern-matcher so that when the rule is applied at run-time, pattern-matching will commence in state K at position Q.

For instance, consider the template *hbyagxa* of rule 3 in the system (E) of Section 4. While trying to match the whole template to the pattern of rule 5, the subterm *gxa* is encountered. The adaptive rewrite strategy requires this to be rewritten if possible. When it is analysed, rule 4 can be applied, and so *gxa* is rewritten to *x*. The strategy now requires the resulting term *hbyax* to be rewritten, if possible. However, pattern-matching halts at the last symbol because it is not known whether the value of *x* will match *a* as in the pattern of rule 5, or not. So *hbya·x* is returned as the new template. The analysis recognises the first four symbols (they match those of the pattern of rule 5), and so pattern-matching can safely

commence at the fifth symbol at run-time. Now, let the subject term be *fabaabaa*. In the original equation system (*E*), the evaluation would proceed by

$$\underline{fabaabaa} \xrightarrow{3} hbaagaa \xrightarrow{4} \underline{hbaaa} \xrightarrow{5} a$$

where redexes are underlined. However, when the template of rule 3 is replaced by *hbya·x* to give a rule 3' the evaluation would proceed by

$$\underline{fabaabaa} \xrightarrow{3'} \underline{hbaa{\cdot}a} \xrightarrow{5} a$$

thereby skipping one rewriting step and some pattern-matching.

5.1. Template pattern-matching

Pattern-matching of templates or their subterms differs from that earlier on because we may now have variable symbols in the subject term. Functions are treated in the same way as before. However, when a variable is encountered at position *P*, there are three possible courses of action. First, suppose that there is only an ω-transition from the current matching state in the matching automaton. (So every pattern in the current matching set has a variable symbol at *P*, and the matching table contains *fail* under each function symbol.) Then, the variable in the pattern can be instantiated to the run-time value of the variable in the subject term. So pattern-matching will succeed at run-time, and analysis can continue as determined by the matching table entry. Secondly, suppose every transition from the current matching state is either *fail* or is identical to the entry under ω in the matching table. Again at run-time, the *non-fail* action will be performed anyway (whether via a function or ω symbol entry). Thus, matching will again succeed at *P*, and so the analysis may proceed as before. Finally, and otherwise, for some function symbol *f* there is a (*non-fail*) *f*-transition from the current matching state that is different from the ω-transition. In this case, the analysis must halt and return *P* paired with the current state because at run-time, the template variable may either match *f* or default to ω, so that the next state is not determined.

The order of analysing the right sides depends upon the choice of reduction strategy. Clearly, after applying a given rule the whole template or one of its subterms, or even a superterm of the template, may be the next candidate for rewriting. The analysis *must* use this strategy to determine the next term to consider. Therefore, a good choice of the reduction strategy will enable more to be gained from template knowledge, more compile-time evaluation to be done, and hence lead to more efficient programs. In particular, this is the case for the adaptive strategy of Section 4. Whenever a term is rewritten, a further attempt can be made to rewrite the result: a template is always the next candidate term to be rewritten, and the right side analysis can therefore consider at least this term.

So, we will present the analysis process using that adaptive strategy on the example above, and assume a constructor discipline as before. In general, the analysis proceeds by inspecting in pre-order the nodes of the template graph starting from its root, i.e. taking the symbols in left-to-right order. The presence of variable symbols in the subject term means that the analysis transition rule (see Figure 5) is a version of that of Figure 3 which is modified in the way described above. A further modification is necessary because the run-time subject term is not known, and the strategy applies differently to subterms, which the template may represent.

$\langle$I, P, MT, R, $\tau\rangle \rightarrow$

Case *MatchingAction* **of**

accept-symbol$_J$: $\langle$J, *Next*(P), MT, R, $\tau\rangle$;

accept-term$_J$: **If** $\forall f \in F_\pi$ (MT[I, *f*] = fail) **then** $\langle$J, *NextArg*(P), MT, R, $\tau\rangle$
else *MarkAsHeadNormal*(τ);

reduce$_r$: **If** τ[R.P]$\in F_\pi$ **or** $\forall f \in F_\pi$ (MT[I, *f*] = fail **or** reduce$_\tau$) **then**
$\langle 0, \Lambda,$ MT, R, *ApplyRule*$_r(\tau,$ R)$\rangle$;
else *MarkAsHeadNormal*(τ);

fail: **If** $\exists f \in F_\pi$(MT[I, *f*] $\neq$ fail) **and** τ[R.P]$\in F_\pi$ **and** $\neg$*IsHeadNormal*(P) **then**
HeadNormalise(τ/R.P)
else Case MT[I,ω] **of**
accept-term$_J$: $\langle$J, *NextArg*(P), MT, R, $\tau\rangle$;
reduce$_r$: $\langle 0, \Lambda,$ MT, R, *ApplyRule*$_r(\tau,$ R)$\rangle$;
fail: *MarkAsHeadNormal*(τ);
End

End

Where *MatchingAction* = **If** τ[R.P] $\in F_\pi$ **then** MT[I, τ[R.P]] **else** fail

Figure 5. *Machine state transition rule for template analysis*

In general, there are three possible outcomes at each pattern-matching step: *failure*, *halting* and *success*. The process halts when run-time knowledge of the value of a variable is required, or when the run-time context of the template is required before the strategy can determine the next candidate term for rewriting. Success leads to a rewrite and the strategy seeks the next redex, bearing in mind that the new template may not be the whole of the subject term being evaluated. If analysis of the whole template produces *failure*, then no rewrite of the current term is possible, and another redex is sought, as in the case of success. For the strategy here, once the current template as a whole returns fail, the process does not try to analyse the subterms. This is because it is not known whether the whole template, one of its subterms or another subterm of the subject term, may become the next candidate to rewrite. Thus, until some variables are instantiated, the template obtained by the analysis is root-redex free, and is therefore said to be in *head* normal form [3,7].

5.2. Transition rules for template analysis

The analysis transition rule for the adaptive strategy is given in Figure 5. The head normal form of a template τ is obtained by repeating that transition rule, starting from the initial state $(0, \Lambda, MT, \Lambda, \tau)$ until it marks τ as head normal. Initially, all the nodes in the graph of τ, except those labelled with constructors, are marked as not head normal (i.e. not known to be in head normal form). Subterms with a constructor root are already in head normal form, and so are marked as such. As before, *MatchingAction* is either $MT[I, \tau[R.P]]$ or *fail*, depending on whether $\tau[R.P]$ is a symbol that appears in the patterns or not. Notice that $\tau[R.P]$ may be the symbol ω. Therefore, we must follow $MT[I, \omega]$, only if there exists no function symbol f of F_π such that $MT[I, f] \neq fail$.

Table II. Decorated matching table for the equation set (E)

	f	g	h	a	b	ω
0	*accept-symbol*$_1$	*accept-symbol*$_2$	*accept-symbol*$_3$	*fail*	*fail*	*fail*
1	*fail*	*fail*	*fail*	"	"	*accept-term*$_4$
2	"	"	"	"	"	*accept-term*$_5$
3	"	"	"	"	*accept-symbol*$_7$	*accept-term*$_6$
4	"	"	"	*accept-symbol*$_{10}$	*accept-symbol*$_8$	*accept-term*$_9$
5	"	"	"	*reduce*$_{4,\langle 0,\Lambda\rangle}$	*fail*	*fail*
6	"	"	"	*reduce*$_{6,\langle 5,2\rangle}$	"	"
7	"	"	"	*fail*	"	*accept-term*$_1$
8	"	"	"	"	"	*accept-term*$_2$
9	"	"	"	*reduce*$_{2,\langle 0,\Lambda\rangle}$	"	*fail*
10	"	"	"	*reduce*$_{1,\langle 0,\Lambda\rangle}$	"	*reduce*$_{1,\langle 0,\Lambda\rangle}$
11	"	"	"	*accept-symbol*$_{14}$	"	*accept-term*$_{13}$
12	"	"	"	*fail*	"	*accept-term*$_{15}$
13	"	"	"	*reduce*$_{6,\langle 5,2\rangle}$	"	*fail*
14	"	"	"	*reduce*$_{5,\langle 0,\Lambda\rangle}$	"	"
15	"	"	"	*reduce*$_{2,\langle 0,\Lambda\rangle}$	*accept-symbol*$_{16}$	*accept-term*$_{16}$
16	"	"	"	*fail*	*fail*	*accept-term*$_{17}$
17	"	"	"	"	"	*reduce*$_{3',\langle 14,2\rangle}$

Eventually, after repeatedly applying the transition of Figure 5, the machine halts in $\langle I, P, MT, R, \tau\rangle$ with τ marked as head normal. Then the symbol $\tau[P]$ or the context of τ is not defined enough to progress in the matching of τ. In both cases, the decoration returned is merely the value of the pair $\langle I, P\rangle$ at this final stage. The repetition of the transition rule of Figure 5 until it halts is called *HeadNormalise*(τ), and it may need to call itself recursively for subterms. The Boolean function *IsHeadNormal*(P) checks whether the subterm rooted at position P is known to be in head normal form, whereas the function *MarkAsHeadNormal*(τ) marks the root of the template τ when this template is known to be in head normal form.

Overall, analysis of all right sides of the equation system (E) yields a revised template list (a, a, $hb\omega a\omega$, ω, a, $ga\omega$), in which the third has changed, and a corresponding list of decorations ($\langle 0, \Lambda\rangle$, $\langle 0, \Lambda\rangle$, $\langle 14, 2\rangle$, $\langle 0, \Lambda\rangle$, $\langle 0, \Lambda\rangle$, $\langle 5, 2\rangle$). The new matching table is given in Table II, complete with decorations. Four of the decorations are $\langle 0, \Lambda\rangle$, which indicate that no progress at all was made in matching the templates.

The analysis of the template $hb\omega ag\omega a$ of rule 3 is described in Table III. This is done using the transition rule of Figure 5 and the original matching table which is the same as the undecorated version of Table II in this instance.

5.3. New machine state transition rule

Using the decorated matching table, together with the new template set, it is now possible to skip all the *known* nodes in a template and avoid all those reductions that have been done at compile-time while analysing the templates. Whenever a rule r is used to rewrite a subterm the new template is used and the corresponding decoration $\langle K, Q\rangle$ from MT is used to set the current matching state and matching position. Figure 6 depicts the transition rule for the machine augmented with these decorations.

Table III. Analysis of the template $hb\omega ag\omega a$

Current State I	Current Position P	Current term	Matching Action
0	Λ	$\cdot hb\omega ag\omega a$	*accept-symbol*$_3$
3	1	$h\cdot b\omega ag\omega a$	*accept-symbol*$_7$
7	1.1	$hb\cdot\omega ag\omega a$	*accept-term*$_{11}$
11	1.2	$hb\omega\cdot ag\omega a$	*accept-symbol*$_{14}$
14	2	$hb\omega a\cdot g\omega a$	fail – *HeadNormalise*($g\omega a$)
0	Λ	$\cdot g\omega a$	*accept-symbol*$_2$
2	1	$g\cdot\omega a$	*accept-term*$_5$
5	2	$g\omega\cdot a$	*reduce*$_4$
0	Λ	$\cdot\omega$	fail – *HeadNormalise*(ω)
14	2	$hb\omega a\cdot\omega$	fail – *MarkAsHeadNormal*($hb\omega a\omega$)
14	2	$hb\omega a\cdot\omega$	return $\langle 14,2\rangle$ and $hb\omega a\cdot\omega$

$\langle$I, P, MT, R, T$\rangle \rightarrow$

Case *MatchingAction* **of**

accept-symbol$_J$: $\langle$J, *Next*(P), MT, R, T$\rangle$;

reduce$_{\tau,\langle K, Q\rangle}$: $\langle$K, Q, MT, R, *ApplyRule*$_r$(T, R)$\rangle$;

fail : **If** T[R.P]$\in F_\pi$ **and** $\exists f \in F_\pi$ (MT[I,*f*]$\neq$fail) **and** $\neg$*IsNormal*(P) **then**

Normalise(T/R.P)

else Case MT[I,ω] **of**

accept-term$_J$: $\langle$J, *NextArg*(P), MT, R, T$\rangle$;

reduce$_{\tau,\langle K, Q\rangle}$: $\langle$K, Q, MT, R, *ApplyRule*$_r$(T, R)$\rangle$;

fail: **If** *Next*(*R*) Defined **then** $\langle 0, \Lambda,$ MT, *Next*(R), T$\rangle$

else *MarkAsNormal*(T);

End

End

Where *MatchingAction* = **If** T[R.P] $\in F_\pi$ **then** MT[I, T[R.P]] **else** fail

Figure 6. *New machine state transition rule*

6. EVALUATION

An experimental implementation of the term rewriting machine described in Sections 4 and 5 was built to evaluate the automaton-driven pattern-matcher, which takes advantage of known right side information. The number of matching actions and rewrites, as well as the evaluation times, has been recorded to show the effect on the evaluation process. Two different *toy* equational programs, *Prog*1 and *Prog*2, have been written to illustrate the effect of different degrees of overlap between the templates and the patterns. We also report the evaluation time for some common problems which were also used as benchmarks to evaluate HIPER [14].

In both programs *Prog*1 and *Prog*2, the function arities are as in the equation system (*E*). Program *Prog*1 (with the equation set below) gains little advantage from the right-hand side analysis. It results in few decorations different from the default decoration. The decoration list is ($\langle 0, \Lambda\rangle$, $\langle 5, 2\rangle$, $\langle 13, 2\rangle$, $\langle 7, 2\rangle$, $\langle 15, 3\rangle$, $\langle 0, \Lambda\rangle$, $\langle 0, \Lambda\rangle$, $\langle 0, \Lambda\rangle$) and the templates remain unchanged (they are already in head normal form):

1. $f(x, a, y) = c$
2. $f(x, y, a) = h(a, x)$
3. $f(x, b(s, y), b(t, z)) = h(b(y, a), z)$
4. $g(x, c) = f(b(a, x), x, b(a, x))$
5. $g(x, y) = f(x, b(x, y), h(y, x))$
6. $h(b(x, a), c) = b(a, x)$
7. $h(b(x, a), y) = c$
8. $h(x, a) = b(a, x)$

Table IV. Number of rewrites, accept-symbols and accept-term-operations

	Rewrites		Accept-symbol operations		Accept-term operations	
	Without decorations	With decorations	Without decorations	With decorations	Without decorations	With decorations
*Prog*1	60	60	137	67	153	85
*Prog*2	42	18	132	24	166	36

However, for program *Prog*2 (with the equation set below), more decorations are different from the default decoration. The list is ($\langle 0, \Lambda\rangle$, $\langle 15, 3\rangle$, $\langle 5, 2\rangle$, $\langle 0, \Lambda\rangle$, $\langle 15, 3\rangle$, $\langle 0, \Lambda\rangle$, $\langle 10, 3\rangle$, $\langle 6, 2\rangle$), and some templates are rewritten. Therefore, *Prog*2 should gain more from the analysis:

1. $f(x, a, y) = c$
2. $f(x, y, a) = h(a, a)$
3. $f(x, b(s, y), b(t, z)) = h(b(y, a), z)$
4. $g(x, c) = f(b(a, x), a, b(a, x))$
5. $g(x, y) = f(x, b(x, y), g(y, c))$
6. $h(b(x, a), c) = g(a, c)$
7. $h(b(x, a), y) = f(c, g(x, a), y)$
8. $h(x, a) = g(a, x)$

After the analysis of right side rewrites and matches, the rewrite rules for *Prog*2 become:

1. $fxay = \cdot c$
2'. $fxya = fabaa \cdot c$
3'. $fxbsybtz = hfybyac \cdot z$
4'. $gxc = \cdot c$
5'. $gxy = fxbxy \cdot c$
6'. $hbxac = \cdot c$
7'. $hbxay = fcfxbxac \cdot y$
8. $hxa = ga \cdot x$

The rule number r is replaced with r' if the template was rewritten in the analysis and the matching dot indicates the position where matching can safely begin when the template instance is matched at run-time.

The numbers of rewrites provided in Table IV clearly show that the use of the decorating information did indeed improve the evaluation time, and the gain depends upon how much of the templates are successfully pattern-matched. Moreover, in our rewriting machine there was virtually no run-time overhead caused by using the decorations.

The terms evaluated are rather large (130 symbols for *Prog*1 and 513 for *Prog*2), and used a combination of the rewrite rules provided. It is of little interest to provide them here. Table IV provides the numbers of rewrites, accept-symbol and accept-term operations performed for *Prog*1 and *Prog*2 when the result of the analysis was considered/neglected.

The evaluation times obtained for both programs in those two cases are given in Table V, along with the evaluation times under the OBJ3 system [29]. The timings were taken on a 50 MHz microSPARC I. Notice that under OBJ3, the same (adaptive) strategy has been used to obtain those figures, and the numbers of rewrites performed was identical to those obtained when the decorations were not considered. Of course, the OBJ3 times include type-checking, etc., which is not the case for our implementation, and so are greater. These timings clearly show that for some equational programs, partial evaluation can provide significant run-time efficiency gains.

Table V. Evaluation times

	Evaluation time (sec.)		OBJ3 time (sec.)
	Without decorations	With decorations	
*Prog*1	0.481	0.421	0.562
*Prog*2	0.337	0.149	0.399

Table VI. Evaluation times for miscellaneous benchmarks

	Evaluation time (sec.)		HIPER time (sec.)
	Without decorations	With decorations	
Kbl	0.088	0.079	0.067
Comm	0.130	0.079	0.10
Ring	2.139	2.060	2.83
Groupl	2.487	1.880	2.00

Table VI shows the performance of our machine together with that of HIPER [14] on some common problems. These problems were first used by Christian [14] to evaluate his system HIPER which uses *flatterms* to perform pattern-matching. The *Kbl* is the ordinary three-axiom group completion problem. The *Comm* is the commutator theorem for groups. The *Ring* problem is to show that if $x^2 = x$ in a ring, then the ring is commutative. Finally, the *Groupl* problem derives a complete set of reductions for Highman's single-law axiomatization of groups using division. Times under HIPER are for Sun 4, while times for our implementation are for MicroSPARC I.

7. CONCLUSION

In the first part of this paper, we described a practical method allowing for the compilation of a set of patterns to an equivalent deterministic automaton which does not need any backtracking to pattern-match terms. In contrast with the method described by Gräf [9], our method presented a simple iterative algorithm for closure, and can handle prioritised overlapping patterns. Patterns are compiled into matching tables which are simple, compact and expressive. Where necessary, the textual order meta-rule was used to resolve conflict due to overlapping patterns. However, we explained how any other meta-rule could easily be implemented.

Unlike Gräf [9], we explain how the pattern-matching method is used in the context of a reduction strategy. In fact, we showed that with minor changes to the abstract machine transition rule, any strategy can be accommodated. We outlined the three most popular rewriting strategies, namely the leftmost-innermost, the leftmost-outermost and the adaptive strategy, the last of which respects the semantics of prioritised equation systems. We described the compile-time analysis of rule right-hand sides that could speed-up the evaluation process, although the advantage gained is heavily dependent upon the choice of evaluation strategy. An important consequence of this compile-time analysis is, that the redex graph may not need to be checked completely each time a pattern-match operation is attempted.

Moreover, we explained how the construction of some terms could be avoided using new templates generated by the analysis procedure.

Finally, in Section 6, results from an implementation have been given. These results show a substantial improvement in some evaluation times for the version that includes right-hand side analysis relative to that which does not.

REFERENCES

1. C. M. Hoffman and M. J. O'Donnell, 'Pattern-matching in trees', *J. ACM*, 68-95 (January 1982).
2. C. M. Hoffman and M. J. O'Donnell, 'Programming with equations', *ACM TOPLAS*, pp. 83-112 (January 1982).
3. R. I. Strandh, 'Compiling equational programs into efficient code', *PhD Thesis*, The Johns Hopkins University, 1988.
4. J. R. Kennaway, 'The specificity rule for lazy pattern-matching in ambiguous term rewriting systems', *Proc. European Symposium on Programming*; *LNCS* **432**, Springer-Verlag, 1990, pp. 256-270.
5. Ph. Schnobelen, 'Refined compilation of pattern-matching for functional programming', *Proc. Algebraic and Logic Programming*; *LNCS* **343**, Springer-Verlag, 1988, pp. 233-243.
6. L. Augustsson, 'Compiling pattern-matching', *Proc. Functional Programming Languages on Computer Architecture*; *LNCS* **201**, Springer-Verlag, 1985, pp. 368-381.
7. P. Wadler, 'Efficient compilation of pattern-matching', in S. L. Peyton Jones (ed.), *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.
8. A. J. Field, L. S. Hunt and R. L. While, 'Best-fit pattern-matching for functional languages', Technical Report, Department of Computing, Imperial College, 1988.
9. A. Gräf, 'Left-to-right pattern-matching', *Proc. Rewriting Techniques and Applications*; *LNCS* **488**, Springer-Verlag, 1991, pp. 323-334.
10. C. M. Hoffman, M. J. O'Donnell and R. I. Strandh, 'Implementation of an interpreter for abstract equations', *Software – Practice and Experience*, **15**(12), pp. 1185-1204 (1985).
11. G. Huet and J. J. Levy, 'Computations in orthogonal term rewriting systems', in J. L. Lassez and G. Plotkin (eds.), *Computational logic*: *Essays in honour of Alan Robinson*, 1992, pp. 415-443.
12. M. J. O'Donnell, *Equational Logic as a Programming Language*, Foundations of Computing Series, The MIT Press, 1985.
13. L. Maranget, 'Two techniques for compiling lazy pattern-matching', Research Report 2385, Institut National de Recherche en Informatique et Automatique INRIA, France, 1994.
14. J. Christian, 'Flatterms, discrimination nets and fast term rewriting', *J. Automatic Reasoning*, **10**, pp. 95-113, 1993.
15. A. V. Aho, R. Sethi and J. D. Ulmann, *Compilers*: *Principles, Techniques and Tools*, Addison-Wesley, 1986.
16. S. C. Johnson, 'Yacc Yet another compiler compiler', Computing Science Technical Report 32, AT&T Laboratories, Murray Hill, NJ, 1975.
17. R. M. Burstall, D. B. MacQueen and D. T. Sannella, 'HOPE: an experimental applicative language', *Proc. 1st ACM LISP Conference*, 1980, pp. 218-225.
18. R. Harper, R. Milner and M. Tofte, 'The definition of standard ML', Technical Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.
19. D. A. Turner, 'Miranda: a non-strict functional language with polymorphic types', *Proc. Functional Programming and Computer Architecture*; *LNCS* **201**, Springer-Verlag, 1985, pp. 1-16.
20. L. Augustsson, 'A compiler for Lazy ML', *Proc. Conference on LISP and Functional Programming*, 1984, pp. 218-225.
21. P. Hudak and P. Wadler, 'Report on the functional programming language Haskell', Technical Report YALEU/DCS/RR656, Department of Computer Science, Yale University, 1988.

22. N. Jones, 'Automatic program specialization', in J. Bjorner (ed.), *Partial Evaluation and Mixed Computations*, Elsevier, 1988, pp. 225-282.
23. I. Durand, D. J. Sherman and R. I. Strandh, 'Fine-grain partial evaluation of intermediate code from equational programs', *Bigre J.*, **74**, (1991).
24. I. Durand, D. J. Sherman and R. I. Strandh, 'Optimisation of equational programs using partial evaluation', *Proc. ACM/IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Sigplan*, **26**, pp. 72-81 (1991).
25. D. J. Sherman, 'Run-time and compile-time improvement to equational programs', PhD thesis, The University of Chicago, Illinois, June 1994.
26. S. Thatte, 'On the correspondence between two classes of reduction systems', *Information Processing Letters*, **20**, pp. 83-85 (1985).
27. B. Salinier, 'Simulation de systèmes de réécriture de termes par des systèmes constructeurs', PhD thesis, Université Bordeaux I, 1995.
28. A. Miniussi and D. J. Sherman, 'Squeezing intermediate construction in equational programs', *Proc. Partial Evaluation International Seminar*; *LNCS* **1110**, Springer-Verlag, 1996, pp. 284-302.
29. J. A. Goguen and T. Winkler, 'Introducing OBJ3', Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, August 1988.

