

Pattern-Matching Without Backtracking

N. Nedjah*
M. Sellami*
S. Eldridge**
C. Walter**

Résumé

Nous proposons une technique pratique pour compiler les fonctions définies par filtrage dans les langages équationnels, en une table de filtrage à partir de laquelle un code efficace peut être dérivé. Nous assumons que l'évaluation est effectuée selon la stratégie normale. En premier lieu, l'ensemble des patterns est converti en un ensemble fermé de patterns qui permet d'effectuer le processus de filtrage sans faire de retour-arrière. Ensuite, une table de filtrage est construite en utilisant une méthode de compilation inspirée de celle utilisée dans YACC. La plupart de la discussion assume que l'ensemble de patterns traité est linéaire à gauche (le cas non-linéaire peut être pris en charge par une passe additionnelle suivant l'étape de filtrage).

Mots clés: filtrage, re-écriture, programmation équationnelle, compilation.

Abstract

We propose a practical technique to compile pattern-matching function definitions in equational-like languages, to a matching table from which efficient code can be derived. It is assumed that the evaluation is performed using normal order strategy. First, the original pattern set is converted to a closed pattern set allowing for the pattern-matching process to be performed without backtracking operations. Next, a matching table is constructed using a compilation method inspired from that YACC employs. Most of the discussion assumes that the processed pattern set is left-linear (the non-linear case can be handled by an additional pass following the matching stage).

Key words: pattern-matching, rewriting, equational programming, compilation.

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 [10,11,14]. 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 [12]. The textual order meta-rule will be used here in the case of overlapping patterns. However, 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 [2,5] and the *case-expression* [3,13]. Another kind of representation consists of a *matching tree* (also called the *index tree*) [6,11]. Similarly, pattern-matching definitions have been compiled into a finite matching automaton in [8,10,12,14,7].

The practical method we shall describe in this paper is inspired by the method of automatic generation of parsers [9] which have been used for many years to compile imperative languages. Backtracking in the pattern-matching process, can consume considerable time as well as space. Therefore, prior to the matching table generation, the original pattern set is converted to an equivalent closed pattern set, using the pattern closure operation defined in [4]. After the method of generating the matching table is described, there follows a description of how to interpret such a matching table.

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 $F = \{f, g, h, \dots\}$ is a set of function symbols, $V = \{x, y, z, \dots\}$ is a set of variable symbols and $R = \{\pi_1 \rightarrow t_1, \pi_2 \rightarrow t_2, \dots\}$ is a set of rewrite rules called the *term rewriting system*, where π_i and t_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 $f(\sigma_1, \sigma_2, \dots, \sigma_n)$ 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 k th child of the

root, and the position $p.k$ denotes the k th ($k \leq 1$) 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 $\pi = \{f\omega\omega, f\omega\omega\}$ is an overlapping pattern set because the term $faac$ is an instance of both patterns whereas the set $\pi' = \{f\omega\omega, fc\omega\omega\}$ 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 $\pi = \{f\omega\omega\omega, f\omega\omega\}$ is a non-overlapping pattern set but it is a prefix-overlapping pattern set because the prefix faa of the term $faaa$ is an instance of both prefixes $f\omega\omega$ 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 $\bar{\pi}$ corresponding to a given pattern set π is the set obtained by applying to π the closure operation defined by Gräf [4] 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:

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

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 ω s 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 $\pi = \{f\omega\omega, f\omega\omega\omega, f\omega\omega\omega\omega\omega\}$ can be converted to an equivalent closed pattern set $\bar{\pi}$ using the closure operation computed as follows:

$$\begin{aligned} \bar{\pi} &= f\pi_f = f\{\omega a\omega, \omega\omega a, \omega g\omega\omega g\omega\omega\} \\ &= f\omega\{a\omega, \omega a, g\omega\omega g\omega\omega\} = f\omega\bar{P} \end{aligned}$$

where

$$\bar{P} = a\bar{P}_a \cup \omega\bar{P}_\omega \cup g\bar{P}_g$$

$$\begin{aligned} &= a\{\omega, a\} \cup \omega\{a\} \cup g\{\omega\omega g\omega\omega, \omega\omega a\} \\ &= \{a\omega, aa, \omega a, g\omega\omega g\omega\omega, g\omega\omega a\} \end{aligned}$$

Then, the closed pattern set $\bar{\pi}$ corresponding to π is:

$$\bar{\pi} = \{f\omega a a, f\omega a \omega, f\omega g\omega\omega a, f\omega g\omega\omega g\omega\omega, f\omega\omega a\}$$

It is clear that the new pattern set accepts the same language 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 [4]).

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.

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 $\{fc\omega\omega, fc\omega a\omega\omega\}$. Using the textual order meta-rule, the first pattern must be matched first, if possible. Then the term $fcg\omega a\omega\omega$ 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 $\bar{\pi} = \{fcg\omega a\omega, fcg\omega a\omega\omega, fcg\omega\omega\omega, fc\omega\omega, fc\omega a\omega\omega\}$ allows matching without backtracking. Then the term will match the second pattern.

Definition 2.8: A matching item is a pattern which is split into a prefix and suffix by inserting the matching dot ($\bullet$) at some point.

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

Definition 2.9: 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 $\bullet\pi_i$ s.t. $\pi_i \in \pi$ whereas matching sets containing items of the form $\pi_i\bullet$ are final matching sets. Note that final matching sets can contain only one matching item.

3. Matching Table Generation

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 (Yet Another Compiler Compiler) to generate parsers for LR-languages [1,9].

The closed pattern set to be compiled is considered as the set of right-hand sides of syntactic productions. For this method, they contain no non-terminal symbols. The pattern set compiles to a matching table whose entries are computed using the information provided by the canonical collection of the matching sets which is derived from the pattern set using the following procedure:

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:

$$\delta(I, s) = \{ \alpha s \beta \mid \alpha s \beta \in I \} \cup \{ \alpha s f \omega^\# f \mu \mid \alpha s \omega \mu \in I \text{ and}$$

for some $f \in F$ and term $\mu', \alpha s f \mu' \in I \}$

Notice that the presence of the items $\alpha s \omega \mu$ together with the items $\alpha s f \mu'$ 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 $\alpha s f \omega^\# f \mu$ are added to remove such non-determinism and avoid backtracking. For instance, let $\pi = \{f\omega\alpha, fc\alpha\}$ and let M be the matching set obtained after accepting the root symbol f so $M = \{f\omega\alpha, fc\alpha\} \cup \{f\omega\alpha\}$. The item $f\omega\alpha$ is added because a target term with the prefix fc could match the pattern $f\omega\alpha$ too if the last argument of f were a rather than c . So supplying the instance $fc\alpha$ 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 $f\omega\alpha$ were taken and a symbol c encountered as the last argument of f in the target term. Also, notice that the transition operation thus described implements exactly the main step in the closure operation due to Gräf [4] 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 $\pi = \{f\omega\alpha, f\omega\alpha, f\omega\omega\alpha\}$ is given in Figure 3.1. 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. 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 3.1) and L is the number of function symbols in F . 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:

<i>accept-symbol</i> _J	if J is not a final state and $s \in F$
<i>accept-term</i> _J	if J is not a final state and $s = \omega$
<i>reduce</i> _r	if J is a final state and r is the matched rule (see next §).
<i>fail</i>	Otherwise, i.e. $J = \emptyset$

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 $s \in F$ is the current input symbol and I the current state then action $MT[I, s]$ is performed if it is not *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 $\pi = \{1:faaa, 2:f\omega a, 3:f\omega g\omega\omega g\omega\}$ of Figure 3.1, the added patterns $faaa$ and $f\omega g\omega a$ would be associated respectively with the rule numbers 1 and 2 if the textual order rule were used.

Ambiguity arises for introduced patterns 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 be associated with π_3 also. 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 .

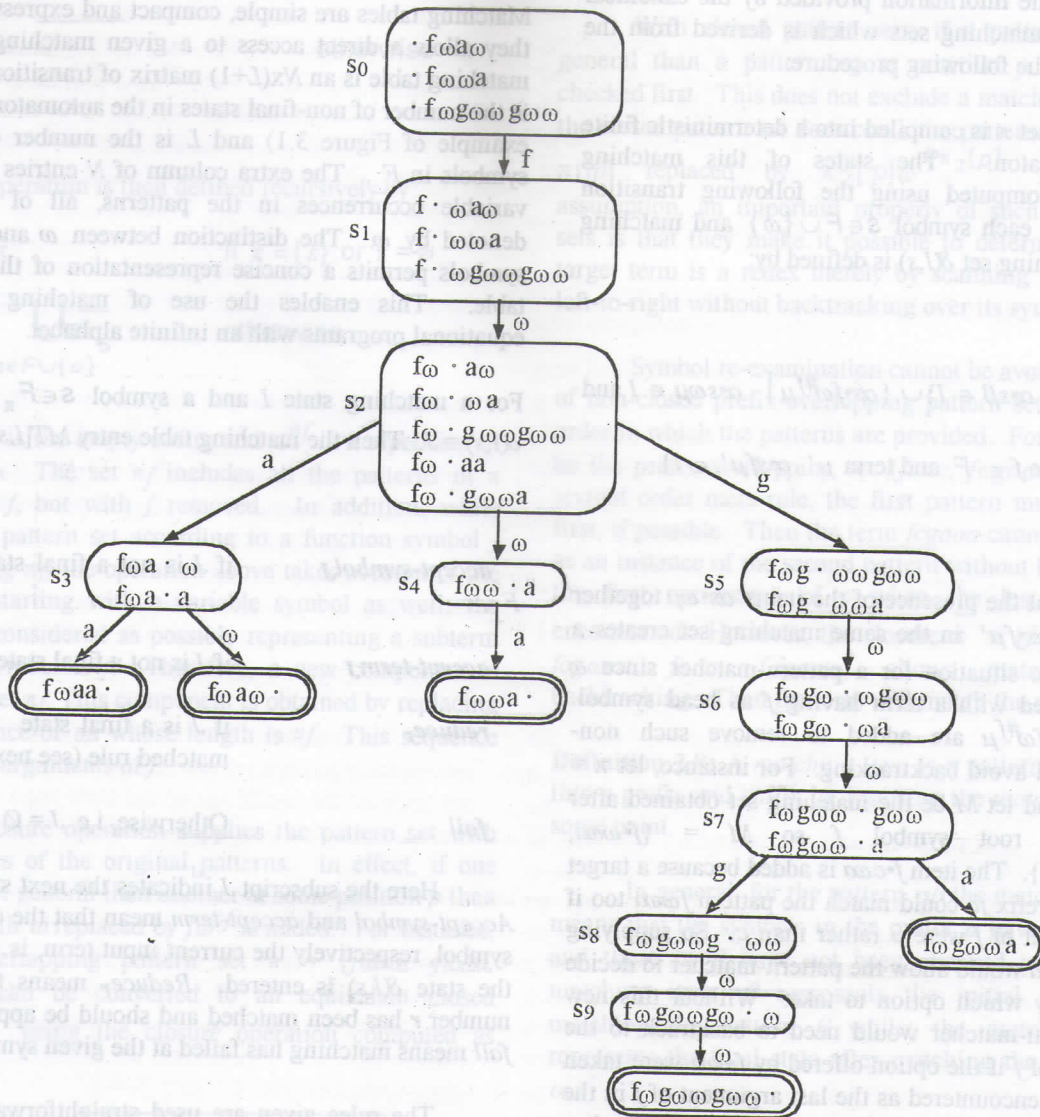


Figure 3.1. Matching automaton for $\omega a \omega, f \omega a, f \omega g \omega \omega g \omega$

The matching table corresponding to the pattern set π of Figure 3.1 is shown in Figure 3.2. 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. Matching Table Interpretation

In this section, the pattern-matching process is viewed as a simple abstract machine represented by the 4-tuple $PM = \langle CS, CP, MT, H \rangle$. CS is the machine current state which corresponds to a row index in the matching table. CP is a pointer to the current node in the subject graph representing the expression to evaluate, held in the heap H . $CP.label$ is used to get the symbol at the current node pointed to by CP . The matching table MT is used to drive the matching process. The initial state of the matching machine is $PM = \langle 0, Root, MT, SubjectGraph \rangle$. Let $\langle CS, CP, MT, H \rangle$ be the current state of the abstract machine, the next state is determined according to $MT[CS, CP.label]$ and $MT[CS, \omega]$ as follows :

accept-symbol_i $\langle i, Next(CP), MT, H \rangle$

accept-term_i $\langle i, NextArg(CP), MT, H \rangle$

reduce_r $\langle 0, Root, MT, Apply-rule_r(H) \rangle$

The subject graph being traversed in preorder, the function $Next(P)$, where P is a graph node pointer, returns the pointer to the next node whereas the $NextArg(P)$ returns a pointer to the next argument. For instance, let *fagbahaa* be the subject term and g the current symbol. Applying the functions $Next$ and $NextArg$ would return b and h respectively. The move performed using the function $NextArg$ allows for a whole term to be accepted. $Root$ represents the pointer to the root of the currently found redex. $Apply-rule_r(P)$ instantiates the template of the matched rule r and replaces the variable nodes by their actual graph-values.

state \ symbol	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

Figure 3.2. Matching table for $\alpha\alpha\omega, f\omega\alpha\alpha, f\omega g\omega\alpha g\omega\omega$

Eventhough the machine state transition rules given above are used in the obvious way, two remarks need be made. First, whenever the entry $MT[CS, CP.label]$ contains *fail*, the entry $MT[CS, \omega]$ is tried. In case its content is *fail*, the heap H is thus holding a graph which does not represent a redex and the flow of control is returned to the caller signalling the end of the evaluation process. Secondly, according to the evaluation strategy in use (applicative order, normal order, ...), the process thus described would invoke itself recursively to find and reduce the next redex.

5. Conclusion

In this paper, we described a practical method allowing for the compilation of a set of patterns to an equivalent deterministic automaton. The generated automaton allows for the pattern-matching process to be performed without any backtracking operation.

Using the matching table, efficient code can be easily generated. Notice that no substitution list has been constructed during the pattern-matching process. As variable occurrences in the left-hand sides are known at compile-time, their corresponding positions can be precomputed and stored. Once a rule has been matched and each time an instantiation of a variable is performed, the corresponding precomputed position is fetched and interpreted according to the used internal representation of terms. The graph representing the variable is then provided to the template instantiating process which uses it to replace the current variable occurrence in the template instance of the matched rule.

Throughout this paper, the textual order meta-rule has been used to solve the conflict due to overlapping patterns. We showed however, how most specific, explicit or other meta-rule can be easily implemented.

Références

- [1] A.V. Aho, R. Sethi, and J.D. Ulmann, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, 1986.
- [2] L. Augustsson, 'Compiling pattern-matching', *Proc. Functional Programming Languages on Computer Architecture*, LNCS 201, pp. 368-381, Springer-Verlag, 1985.
- [3] Field, A.J., Hunt, L.S. and While, R.L., 'Best-fit pattern-matching for functional languages', Technical Report, Department of Computing, Imperial College, 1988.
- [4] A. Gräf, 'Left-to-right pattern-matching', *Proc. Rewriting Techniques and Applications*, LNCS 488, pp. 323-334, Springer-Verlag, 1991.
- [5] R. Heuillard, 'Pattern-matching in trees', LNCS 308, pp. 111-128, 1988.
- [6] C.M. Hoffman and M.J. O'Donnell, 'Pattern-matching in trees', *Journal of ACM*, pp. 68-95, January 1982.
- [7] C.M. Hoffman and M.J. O'Donnell, 'Programming with equations', *ACM TOPLAS*, pp. 83-112, January 1982.
- [8] 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, December 1985.
- [9] S.C. Johnson, 'Yacc - Yet another compiler compiler', *Computing Science Technical Report 32*, AT&T Laboratories, Murray Hill, N.J., 1975.
- [10] J.R. Kennaway, 'The specificity rule for lazy pattern-matching in ambiguous term rewriting systems', *Proc. European Symposium On Programming*, LNCS 433, pp. 256-270, Springer-Verlag, 1990.
- [11] M.J. O'Donnell, *Equational logic as programming language*, *Foundations of Computing Series*, The MIT Press, 1985.
- [12] Ph. Schnobelen, 'Refined compilation of pattern matching for functional programming', *Proc. Algebraic and Logic Programming*, LNCS 343, pp. 233-243, 1988.
- [13] R.I. Strandh, 'Compiling equational programs into efficient code', Ph.D. Thesis, The Johns Hopkins University, 1988.
- [14] P. Wadler, 'Efficient compilation of pattern matching', in S.L. Peyton Jones, *The implementation of functional programming languages*, Prentice Hall International Series in Computer Science, 1987.

* Institut d'Informatique - Université d'Annaba

**UMIST. University Manchester