An Effective Algorithm for Compiling Pattern Matching Keeping Laziness

Pedro Palao and Manuel Núñez
Departamento de Informática y Automática
Universidad Complutense de Madrid
fax: (34-1) 394 4607 ph: (34-1) 394 4468
e-mail: {ecceso,manuelnu}@eucmvx.sim.ucm.es

Abstract

In this paper we present an algorithm for compiling functions defined by pattern matching with side conditions, which can be easily implemented. As previous approaches, this algorithm has a complete set of rules. The generated code has no backtracking and side conditions are evaluated at most once, which represents an advantage with respect to most of the previous algorithms. This algorithm is parameterized by a *subroutine*, which can be chosen such that the result of the compilation fulfills certain properties. In this paper we choose a subroutine that keeps the *laziness* of a list of patterns. But in contrast with previous algorithms (characterization algorithms), this algorithm does not previously determine if the pattern is lazy or not; our algorithm works with any kind of patterns, generating a *lazy* code if the pattern is lazy and even finding the lazy subpatterns of a non lazy pattern. A fundamental concept in order to apply this subroutine is the concept of *distinguisher* of a pattern, which indicates if a *column* of a list of patterns must be chosen to *expand* with it the algorithm.

Keywords: Functional Programming, Pattern Matching, Laziness, Compilation.

1 Introduction

Pattern Matching has been widely studied in the theory of Term Rewriting. This problem can be stated as: given a list of terms $p_1, \ldots, p_n$ and a term t, find whether t is an instance of any of the p_i. The most straightforward algorithm to solve this problem is checking t against each p_i, but this solution is not acceptable because the running time depends on the number of terms in the list. Several algorithms have been developed to solve this problem more efficiently (see [HO82], [Grä91]).

In this paper we restrict ourselves to the study of pattern matching in the implementation of functional programming languages. In functional programming, this problem has some specific features; for example, it is usual to add some strategy in order to decide which of the possible p_i such that t is an instance of p_i is chosen (usually the first top–down). There are also specific algorithms (see [Aug85], [Wad87], [Lav88], [Sch88]) for functional programming.

But previous algorithms usually do not deal with *laziness*. Intuitively speaking, laziness means that a value is only computed when it is needed in order to evaluate an expression. In order to know whether a value matches a pattern, this value must be evaluated to *head normal form*. Then, a lazy language must take care about how

pattern matching is performed, evaluating arguments as less as possible to determine if the argument matches the pattern. Nevertheless, almost all the implementations of recent functional languages do not consider techniques which perform pattern matching in a lazy way.

In this paper we propose an algorithm with the following features:

- Each of the arguments is parsed at most once in order to determine which pattern matches (therefore, there is no backtracking in the compiled code).
- Side conditions, which may be very hard to evaluate, are just tried at most once, and only when it is not possible to distinguish by patterns. Previous algorithms usually do not deal with side conditions.
- One expects, that in a lazy language, the order of evaluation over the argument structure is performed such that the function may diverge (at this point) only if it diverges with any order of evaluation. Our algorithm deals with this topic (laziness), but in contrast with other algorithms ([Lav87, Lav88]), it does not previously characterize if the pattern is lazy or not (see definition 4).

The remainder of the paper is organized as follows. Section 2 introduces preliminary definitions. Section 3 gives the bulk of our algorithm for compiling functions defined by pattern matching. In Section 4 we introduce a subroutine, that combined with the previous algorithm, keeps laziness. In Section 5 we give some outlines for the implementation of the algorithm. Finally, Section 6 presents our conclusions.

2 Definitions

Definition 1 Let Σ be a finite ranked alphabet which is the disjoint union of alphabets Σ_n ($\Sigma = \biguplus_{n\in\mathbf{N}} \Sigma_n$). We consider a set of variable symbols Σ_X such that $\Sigma_X \subseteq \Sigma_0$. $\alpha \in \Sigma_n$ means that α has *arity* n. The set of Σ-terms is defined inductively as the least set such that if $\alpha \in \Sigma_n$, and $t_1, t_2, \ldots, t_n \in \Sigma$-terms, then $\alpha\, t_1\, t_2\, \cdots\, t_n \in \Sigma$-terms. Given a Σ–term $t = \alpha\, t_1\, t_2\, \cdots\, t_n$, we will denote t_i by $t[i]$ (for $1 \leq i \leq n$), and α by $t[0]$. Given $x \in \Sigma_X$, x is denoted by $x[0]$, while $x[i]$ ($i > 0$) denote fresh variables.

Definition 2 A *pattern* is a tuple of Σ-terms. We say that a pattern is *linear* if there is no variable which appears twice in the same pattern. An *instance* of a pattern is a tuple of terms which can be obtained from the pattern by replacing all the variables by any values. Let σ be a function which maps variables into terms. We call *substitution* the extension of σ as a morphism from terms to terms.

Note that if a term is an instance of a pattern, then there exists a substitution σ which yields the term as image of the pattern. From now on, when we consider lists of patterns, we will suppose that all the patterns (tuples) have the same length.

Definition 3 Let $[p_1, \ldots, p_m]$ be a list of patterns. We say that a tuple of terms t *matches* p_i, if p_i is the first pattern of the list such that t is an instance of p_i. We say that an algorithm that decides if t matches p_i exploring t (starting from the root of each argument in t, and comparing the symbols encountered with the symbols in the corresponding part of the pattern) is a *matching strategy*. Note that, with this definition, there is at most one p_i in P such that t matches p_i.

Definition 4 We say that a matching strategy $\mathcal{E}$ for a list of patterns P is *lazy* if for any t such that $\mathcal{E}$ diverges exploring t, then any other strategy diverges exploring t. We say that a list of patterns P is *lazy* if there is a lazy strategy for P.

Example 1 Let f and g be the functions that follow:

$$\begin{array}{llll} f & [\,] & [\,] & = exp_1 \\ f & x & y & = exp_2 \end{array} \qquad \begin{array}{llll} g & [\,] & [\,] & = exp_1 \\ g & x & y{:}ys & = exp_2 \end{array}$$

There is no lazy strategy for $P = [([\,],[\,]),\ (x,y)]$. Consider the call $f\ (a_1{:}a_2)\ \Omega$ (where Ω is a divergent argument). If the evaluation starts with the first argument, the value exp_2 is obtained, while if the evaluation starts with the second argument, a divergent computation is produced. But $f\ \Omega\ (a_1{:}a_2)$ diverges starting with the first argument, while starting with the second one, returns exp_2.

On the other hand, there is a lazy strategy for $Q = [([\,],[\,]),(x,y{:}ys)]$: the second argument is evaluated before the first one. Nevertheless, lazy functional languages like Gofer or Miranda give a divergent computation in the evaluation of $g\ \Omega\ (a_1{:}a_2)$.

Definition 5 We say that a function is *defined by pattern* if

- Its definition is a list of triples (pattern, expression, side condition).
- Its value, when applied to an argument t, is obtained finding the triple (p_i, exp_i, con_i) such that t matches p_i by a substitution σ and con_i holds for this substitution, and then evaluating the result of applying σ to exp_i. If there is no such a p_i, then an error message is produced.

Definition 6 Let P be a list of patterns $[p_1,\ldots,p_m]$, where $p_j = (p_{j1},\ldots,p_{jn})$, and let $q = (q_1,\ldots,q_k)$ be a pattern. The next notation will be used:

$$\begin{array}{rcl} [p_i]_{i=1}^{n} & = & [p_1,\ldots,p_n] \\ q{\updownarrow}i & = & q_i \\ q\bar{\updownarrow}i & = & (q_1,\ldots,q_{i-1},q_{i+1},\ldots,q_k) \\ P{\updownarrow}i & = & (p_1{\updownarrow}i,\ldots,p_n{\updownarrow}i)\ \text{(column } i\text{)} \\ P\bar{\updownarrow}i & = & [p_1\bar{\updownarrow}i,\ldots,p_n\bar{\updownarrow}i] \end{array} \qquad \begin{array}{rcl} \#Var_i & = & \text{Number of variables in } P{\updownarrow}i \\ \#C_i & = & \text{Number of } \textit{root} \text{ occurrences} \\ & & \text{of the constructor } C \text{ in } P{\updownarrow}i \\ & & \\ Cons_i & = & \{C_i \mid \#C_i \neq 0\} \end{array}$$

Definition 7 Let $P = [(p_{k1},\ldots,p_{kn})]_{k=1}^{m}$. We define the *constructor selection function* for a constructor C in column i ($1 \leq i \leq n$), denoted by γ, as the function which satisfies the following conditions:

- $\gamma{:}\ 1..(\#C_i + \#Var_i) \to 1..m$
- $\gamma(j) < \gamma(j+1),\ \forall j(1 \leq j < \#C_i + \#Var_i)$
- $p_{\gamma(j)\,i}[0] \in \Sigma_X \cup \{C\},\ \forall j(1 \leq j \leq \#C_i + \#Var_i)$, i.e. $p_{\gamma(j)\,i}$ is either a variable or a term $t = C\,t_1 \cdots t_k$.

Lemma 1 Given a list of patterns P, a constructor C and a column i, the definition of γ for the constructor C in column i is unique.

Intuitively speaking, given a column and a constructor C, γ considers the terms which either have the constructor C in the root or are variables, preserving the ordering in the list of patterns.

Example 2 Let $P = [(x{:}xs, y{:}ys), (x{:}xs, y{:}ys), (xs, [\,]), ([\,], ys)]$. Then, for ":" and the column 2, we have: $\gamma{:}\, 1..3 \to 1..4$, and $\gamma(1) = 1,\ \gamma(2) = 2,\ \gamma(3) = 4$. For "$[\,]$" and the column 2, we have: $\gamma{:}\, 1..2 \to 1..4$, and $\gamma(1) = 3,\ \gamma(2) = 4$.

Definition 8 Let P be a list of patterns $[(p_{k1}, \ldots, p_{kn})]_{k=1}^{m}$. We define the *variable selection function* for a column i $(1 \leq i \leq n)$, denoted by γ^X, as the function which satisfies the following conditions:

- $\gamma^X{:}\, 1..\#\mathit{Var}_i \to 1..m$
- $\gamma^X(j) < \gamma^X(j+1),\ \forall j(1 \leq j < \#\mathit{Var}_i)$
- $p_{\gamma^X(j)\,i}$ is a variable, $\forall j(1 \leq j \leq \#\mathit{Var}_i)$.

Lemma 2 Given a list of patterns P and a column i, the definition of γ^X for the column i is unique.

Definition 9 Let $P = [(p_{k1}, \ldots, p_{kn})]_{k=1}^{m}$ be a list of patterns and $t = (t_1, \ldots, t_n)$ be a tuple of terms. We define the *subpattern of P generated by t*, denoted by $\mathcal{S}_t(P)$, as the list of patterns defined as:

1. If $\forall i(t_i \in \Sigma_X)$, then $\mathcal{S}_t(P) = P$.
2. If $\exists j(t_j \notin \Sigma_X)$, then $\mathcal{S}_t(P) = \mathcal{S}_{t'}(P')$ where $C = t_j[0]$, r is the arity of C, $s = \#\mathit{Var}_j + \#C_j$, $t' = (t_1, \ldots, t_{j-1}, t_j[1], \ldots, t_j[r], t_{j+1}, t_n)$, and $P' = [(p_{\gamma(k)\,1}, \ldots, p_{\gamma(k)\,j-1}, p_{\gamma(k)\,j}[1], \ldots, p_{\gamma(k)\,j}[r], p_{\gamma(k)\,j+1}, \ldots, p_{\gamma(k)\,n})]_{k=1}^{s}$

Lemma 3 Let P be a list of patterns, and t be a term. If $\mathcal{S}_t(P)$ is not a lazy list of patterns then P is not a lazy list of patterns.

3 The Algorithm

In this section we present a formal description of our algorithm. Our algorithm has a function defined by pattern as argument and returns an expression which, considered as a tree, has in the internal nodes a **case** clause over a simple pattern or an **if** clause over any of the side conditions. The *leaves* of the tree are the expressions that define the function.

Although the algorithm only works with linear patterns, it can be generalized to non linear patterns in the usual way, changing the repeated variables for new ones and adding an equality condition to the side condition.

We will specify the algorithm as a function *compile* which has a function defined by pattern as argument and returns the tree expression.

Definition 10 Let $f = [((v_{k1}, \ldots, v_{kn}), exp_k, con_k)]_{k=1}^{m}$ be a function defined by pattern. We define the function *compile* as

$$compile\ f = match\ (u_1, \ldots, u_n)\ f$$

where $u_1, \ldots, u_n$ are fresh variables indicating the length of the patterns $\bar{v}_k$.

The rest of the section is devoted to define the function *match*. This function is inductively defined, with the property that in recursive calls, the ordering of the triples in the original definition is preserved. We give a complete set of rules; some of them are similar to those in [Wad87] (Empty Rule, Variable-Column Rule) while others are specific for our algorithm.

3.1 Base Rules

There are two base cases: when the list of variables is empty (first argument), and when the list of triples is empty (second argument).

When the list of variables is empty, all the expressions are equally acceptable, because there are no patterns. We use the side condition in order to know which of the expressions is chosen. The algorithm must keep the order in the function, and for this reason this case has to be compiled with a sequence **if** $\cdots$ **elsif** $\cdots$ **else**, finishing with a *failure* clause (used if no condition evaluates to true).

Rule 1 (Empty Rule)

$match\ (\ )[((\ ), exp_1, con_1), ((\ ), exp_2, con_2), \cdots, ((\ ), exp_m, con_m)] =$

if con_1 **then** exp_1 **elsif** con_2 **then** exp_2 $\cdots$ **elsif** con_m **then** exp_m **else No Match**

Another base case appears when the list which defines the function is empty. That means that the pattern is not exhaustive, and a run–time error must be produced. This error is not a fault with a *backtracking jump* like in [Aug85], but indicates that the function argument matches no pattern (considering side conditions) in the definition of the function.

Rule 2 (Fail Rule) $match\ (u_1, \ldots, u_n)\ [\,] =$ **No Match**

3.2 Inductive Rules

Now we consider the inductive cases. There are two rules which simplify the call to *match*, and another rule which is used if none of the previous rules can be applied (Default Rule).

The first rule can be applied when the first triple has a pattern only with variables. Then, an **if** is generated with the condition con_1, substituting the variables appearing in the pattern with the values $(u_1, \ldots, u_n)$, the expression exp_1 in the **then** part (applying the same substitution) and, in the **else** part, the result of the rest of compilation.

Rule 3 (Variable-Row Rule) If $v_{11}, \ldots, v_{1n}$ are variables, then
$match\ (u_1, \ldots, u_n)[((v_{k1}, \ldots, v_{kn}), exp_k, con_k)]_{k=1}^{m}\ =$

$$\textbf{if } con_1[u_1/v_{11}, \ldots, u_n/v_{1n}] \ \textbf{ then } exp_1[u_1/v_{11}, \ldots, u_n/v_{1n}]$$
$$\textbf{else } match\ (u_1, \ldots, u_n)[((v_{k1}, \ldots, v_{kn}), exp_k, con_k)]_{k=2}^{m}$$

The second rule is applied if there exists a column in the list of patterns such that only variables appears in this column. This can be solved by removing this variable, and then, performing a renaming both in the expressions and in the side conditions.

Rule 4 (Variable-Column Rule) If there exists a *column i* such that all the terms are variables, then

$$match\ (u_1, \ldots, u_i, \ldots, u_n)[((v_{k1}, \ldots, v_{ki}, \ldots, v_{kn}), exp_k, con_k)]_{k=1}^{m} =$$
$$match\ (u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n)$$
$$[((v_{k1}, \ldots, v_{k\,i-1}, v_{k\,i+1}, \ldots, v_{kn}), exp_k[u_i/v_{ki}], con_k[u_i/v_{ki}])]_{k=1}^{m}$$

If none of the previous rules can be applied, then the *Default Rule* is applied. This rule expands with a column (the algorithm which chooses this column will be presented in section 4). Intuitively speaking, if the i-th argument of the function is evaluated enough to find a constructor in the root, then we can discard most of the patterns of the i-th column. We only have to consider the patterns which have the previously obtained constructor in the root, and the patterns which are variables. In the former case, the subexpressions of the argument still have to be compared with the constructor arguments, but in the latter this comparison is not necessary; in these triples, new variables are needed (according to the arity of the considered constructor). This is shown in the following

Example 3 Consider the call to the *match* function

$$\begin{array}{lll}
match\ (u_1, u_2) & & \\
\quad [((x{:}xs,\ y{:}ys), & x : \text{merge } xs\ (y{:}ys), & x \leq y), \\
\quad\ ((x{:}xs,\ y{:}ys), & y : \text{merge } (x{:}xs)\ ys, & x > y), \\
\quad\ ((\ xs, \quad [\,]\ ), & xs, & true\ ), \\
\quad\ ((\ [\,], \quad ys\ ), & ys, & true\ )]
\end{array}$$

and suppose that the second column is chosen to expand with. The case expression has two entries: one for "[ ]" and another one for ":". In the first entry, the third triple (since it has "[ ]" in the root) and the fourth one (because it is a variable) are placed. The first, second and fourth ones appear in the second entry. The result is

$$\begin{array}{lll}
\textbf{case } u_2 \textbf{ of} & & \\
\quad [\,] \quad \Rightarrow & match\ (u_1) & \\
 & \quad [((xs), \ xs, \ true), & \\
 & \quad\ (([\,]), \ u_2, \ true)] & \\
\quad w_1{:}w_2 \quad \Rightarrow & match\ (u_1, w_1, w_2) & \\
 & \quad [((x{:}xs,\ y,\ ys), & x : \text{merge } xs\ (y{:}ys), \quad x \leq y), \\
 & \quad\ ((x{:}xs,\ y,\ ys), & y : \text{merge } (x{:}xs)\ ys, \quad x > y), \\
 & \quad\ ((\ [\,],\ \alpha_1{,}\alpha_2), & u_2, \qquad true\ )]
\end{array}$$

where $w_1, w_2, \alpha_1, \alpha_2$ are fresh variables.

Rule 5 (Default rule) If the column i is chosen to expand with, then

$$
\begin{array}{l}
match\ (u_1,\ldots,u_i,\ldots,u_n)[((v_{k1},\ldots,v_{ki},\ldots,v_{kn}),exp_k,con_k)]_{k=1}^{m} = \\
\textbf{case}\ u_i\ \textbf{of} \\
\quad C^1\bar{w}_1 \quad \Rightarrow match\ (u_1,\ldots,u_{i-1},u_{i+1},\ldots,u_n) +\!\!+ \bar{w}_1 \\
\qquad\qquad\qquad\qquad [(v_{\gamma_k^1}\bar{\updownarrow}i +\!\!+ (v_{\gamma_k^1\,i}[1],\ldots,v_{\gamma_k^1\,i}[n_1]),exp'_{\gamma_k^1},con'_{\gamma_k^1})]_{k=1}^{s^1} \\
\quad \ldots \qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
\quad \ldots \qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
\quad C^r\bar{w}_r \quad \Rightarrow match\ (u_1,\ldots,u_{i-1},u_{i+1},\ldots,u_n) +\!\!+ \bar{w}_r \\
\qquad\qquad\qquad\qquad [(v_{\gamma_k^r}\bar{\updownarrow}i +\!\!+ (v_{\gamma_k^r\,i}[1],\ldots,v_{\gamma_k^r\,i}[n_r]),exp'_{\gamma_k^r},con'_{\gamma_k^r})]_{k=1}^{s^r} \\
\textbf{otherwise} \Rightarrow match\ (u_1,\ldots,u_{i-1},u_{i+1},\ldots,u_n) \\
\qquad\qquad\qquad\qquad [(v_{\gamma_k^x}\bar{\updownarrow}i, exp_{\gamma_k^x}[u_i/v_{\gamma_k^x\,i}], con_{\gamma_k^x}[u_i/v_{\gamma_k^x\,i}])]_{k=1}^{s^x}
\end{array}
$$

$$
\text{where}\ \left\{\begin{array}{rcl}
Cons_i & = & \{C^1,C^2,\ldots,C^r\},\ s^x = \#Var_i,\ s^a = s^x + \#C_i^a\ \ (1\le a\le r) \\
\gamma_b^a & = & \gamma(b),\ \text{where } \gamma \text{ is the selection function for } C^a \text{ in column } i \\
\gamma_b^x & = & \gamma^X(b),\ \text{where } \gamma^X \text{ is the variable selection function for column } i \\
n_a & = & \text{arity of the constructor } C^a\ \ (1\le a\le r) \\
exp'_l & = & \left\{\begin{array}{ll} exp_l & ,\ \text{if } \neg Var(v_{l\,i}) \\ exp_l[u_i/v_{l\,i}] & ,\ \text{otherwise}\end{array}\right. \\
con'_l & = & \left\{\begin{array}{ll} con_l & ,\ \text{if } \neg Var(v_{l\,i}) \\ con_l[u_i/v_{l\,i}] & ,\ \text{otherwise}\end{array}\right.
\end{array}\right.
$$

and for any a, b such that $1 \le a \le r$, $1 \le b \le s^a$, and such that $v_{\gamma_b^a\,i}[0] \in \Sigma_X$ (i.e. $v_{\gamma_b^a\,i}$ is a variable), $v_{\gamma_b^a\,i}[1],\ldots,v_{\gamma_b^a\,i}[n_a]$ are fresh variables. With this condition we ensure that patterns remain linear and that there is no name capture.

Note that if any v_{ki} is a variable, then there is an occurrence corresponding to the k-th triple, for every entry $C^j\bar{w}_j$, and another one in the **otherwise** clause. The **otherwise** clause is used for the constructors which do not appear explicitly in the **case** expression. For this reason, if all the constructors of the type of the column i are in $Cons_i$ (i.e. column i is *exhaustive*), the **otherwise** clause may be removed (as in Example 3). A consequence of the Default Rule is that one expression may appear in different places. In Section 5, we show how a code without repetitions can be generated.

The compilation to a virtual machine of the **case** expression is usually done using a table of memory directions, which is indexed by the type constructors.

4 Choosing a *good* column

In the previous section, we have not given an algorithm which selects a column to expand with. Now, we give an algorithm that selects these columns keeping laziness. The idea is to find the columns which must be (necessarily) evaluated to determine if a term matches a pattern. Next, we introduce the concept of *distinguisher*.

Definition 11 Let $P = [p_1, \ldots, p_m]$ be a list of patterns. We say that p_i with $1 \leq i \leq m+1$ (for convenience p_{m+1} will be a pattern only with variables), is a *distinguisher* of P, if there is a tuple of terms t which is an instance of p_i and such that it is not an instance of any p_j for $1 \leq j < i$. We say that p_i is a *distinguisher* of P *for the column* j if $p_i \bar{\updownarrow} j$ is a distinguisher of $P \bar{\updownarrow} j$ and p_{ij} is a variable.

Let us remark that there exists a distinguisher for a column which only has constructors in the root iff the rest of the columns are not exhaustive. This distinguisher would be in the *row* $m+1$, because there are no variables in that column.

Lemma 4 Let P be a list of patterns. P has a matching strategy not evaluating t_j in a list of arguments $t = (t_1, \ldots, t_n)$ iff P has a distinguisher for the column j.
Proof: Let t be a list of arguments such that it is not necessary to evaluate t_j, and let p_i be such that t matches p_i. Then, p_{ij} must be a variable; otherwise t_j would have to be evaluated in order to know whether it coincides with the constructor. Also, the fact that t does not match any p_k with $k < i$ implies that $t \bar{\updownarrow} j$ does not match any $p_k \bar{\updownarrow} j$ with $k < i$. Thus, $t \bar{\updownarrow} j$ is the instance that makes p_i a distinguisher of P for the column j.

Now, suppose that p_i is a distinguisher of P for the column j. Then, p_{ij} is a variable and there is an instance t of $p_i \bar{\updownarrow} j$ which is not an instance of $p_k \bar{\updownarrow} j$ (for any k, $1 \leq k < i$). Then, a strategy which checks if an argument coincides with t in every column but in the j-th, is a strategy which does not evaluate the j-th column for the instance t.

Lemma 5 Let P be a lazy list of patterns, and j be a column which has a distinguisher. Then, a lazy strategy cannot expand with column j.
Proof: Let $t = (t_1, \ldots, t_n)$ be a list of terms such that $t \bar{\updownarrow} j$ is the instance with whom p_i is a distinguisher for the column j. Then, there is a strategy $\mathcal{E}$ that does not evaluate the column j when is applied to an argument a such that $a \bar{\updownarrow} j = t \bar{\updownarrow} j$. Thus, this strategy does not diverge when the argument $a = (t_1, \ldots, t_{j-1}, \Omega, t_{j+1}, \ldots, t_n)$ is applied. Easily follows that a lazy strategy does not diverge when the argument a is applied, and thus it can not be possible to expand with the column j.

Definition 12 We define the set of *admissible columns* for a list of patterns P, denoted $\mathcal{A}(\mathcal{P})$, as the set of columns which have not a distinguisher.

Corollary 1 Let P be a lazy list of patterns. Then $\mathcal{A}(\mathcal{P})$ is not empty.

This corollary gives a necessary condition for a list of patterns to be lazy. The following example shows that "$\mathcal{A}(\mathcal{P}) \neq \emptyset \Rightarrow \mathcal{P}$ is a lazy list of patterns" (the reciprocal of Corollary 1) does not hold.

Example 4 Consider $P = [([\,], [\,]), ([\,], y{:}ys), (x{:}xs, [\,]), (x{:}[\,], y{:}[\,]), (x{:}xs, y{:}ys)]$. P is exhaustive and all the terms have constructors in the root. Then, both columns are admissible ones (and thus $\mathcal{A}(\mathcal{P}) \neq \emptyset$). But the subpattern corresponding to the term $(a_1{:}b_1,\ a_2{:}b_2)$ is $[(x, [\,], y, [\,]),\ (x, xs, y, ys)]$, and (as we saw in function f of example 1) there is no lazy strategy for matching (b_1, b_2) with $[([\,], [\,]),\ (xs, ys)]$. By Lemma 3, P is not lazy because it has a subpattern which is not lazy.

Lemma 6 Let P be a lazy list of patterns and let $A = \mathcal{A}(\mathcal{P})$. Then, for any $i \in A$, there is a lazy strategy that expands with the column i.
Proof: P is lazy implies that there exists a lazy strategy $\mathcal{E}$ for it. This strategy must expand with any of the columns in A (Lemma 5). Suppose that $i \in A$ is chosen to expand with, and consider $j \in A$ such that $j \neq i$. We know that $\mathcal{E}$ always has to evaluate the column j. Then, we consider a strategy which *interchanges* the points of evaluation of i and j. Obviously, this strategy is also lazy.

Corollary 2 Let P be a lazy list of patterns. A pattern strategy is lazy iff this strategy expands with a column in $\mathcal{A}(\mathcal{P})$.

After corollary 2, we can give our algorithm to choose a column to expand with.

Algorithm (**Choose a column to expand with**)
Let P be a list of patterns, and let A be the set of admissible columns. If $A \neq \emptyset$ we choose any one in the set A. If $A = \emptyset$ we choose any column (that means that the list of patterns is not lazy).

This algorithm chooses a column keeping laziness (Lemma 6), and if the list of patterns is lazy, this choice gives a lazy strategy (Corollary 2). As we showed in Example 4, the idea of admissible column represents a characterization of *local* laziness over a part of the argument that it is been explored. Thus, our algorithm can isolate the part of the pattern where the problem (no laziness) appears, and it can compile the rest of the pattern in a lazy way.

Example 5 We will show that in Example 3 the second column is chosen by our algorithm. We must show that the second column is an admissible one, while the first one is not. This fact is because there is a distinguisher for the first column in the third row: the instance $([\,])$ gives us the result. After third row, the first column is exhaustive and that is why it can not be any distinguisher for the second one.

Theorem 1 Let P be a lazy list of patterns. Then, the algorithm in the previous section, parameterized with the algorithm above to choose a column, gives a lazy matching strategy.
Proof: The proof is done by induction over the rules of the algorithm. The base rules (Empty Rule and Fail Rule) give a lazy strategy because they do not perform any decision over the list of patterns. The result for the Variable–Row and Variable–Column rules is immediate by induction, since none of them evaluate any argument. The proof of laziness for the Default Rule can be done using Lemma 5 and by induction.

5 Implementation

In this section we deal with some details of the implementation of the algorithm.

5.1 Calculation of the set of admissible columns

First, we suppose that all the columns are admissible, and for each column, we create a set of instances (initially empty). We go top-down over the list of patterns, looking for variables and updating the set of instances of each column with the instances of the rest of the pattern (for the pattern p_i and the column j, the rest of the pattern is $p_i \bar{\updownarrow} j$). If one variable is found in one of the columns, which belongs to the set of admissible columns, we determine if the rest of the pattern is a distinguisher for this column (looking at the associated set). If it is a distinguisher, we remove this column from the set of admissible ones and we forget the associated set (which will not be needed any more). We repeat this process until the end of the list of patterns is reached or the set of admissible columns is empty. The update of the set of instances can be done *by request* instead of doing it for every pattern in the list of patterns; that is, the update is done only when a variable is found.

5.2 Improvements to the efficiency of the algorithm

Increasing the speed of the algorithm can be done when the Default Rule has to be applied. If there is a column which only has constructors, and the rest of the columns are exhaustive, then this column can be chosen keeping laziness, because it will be in the set of admissible columns. This is very effective and it is presented very often; for example, this technique can be applied to all the examples presented in [Lav87, Lav88].

When the set of admissible columns for a list of patterns has been calculated, the problem is which of them is chosen. The next heuristic rule can be applied to locally minimize the number of steps that the algorithm must perform: choose the column which minimizes the number of constructors multiplied by the number of variables. This rule tries to expand as less as possible the patterns which have variables, leaving the problem as small as possible. In a similar way, this heuristic rule can be applied if there is no admissible column. Another technique is to expand with all the admissible columns, and calculate the set of admissible columns for the new list of patterns. With this technique, we get that some calculations, which would be done several times, are done only once.

5.3 Duplication of code

Obviously, the code duplication is solved, sharing the expression among the different *branches* in the **case** tree where the same code appears. Let T be a triple (*patt*, *exp*, *con*), such that one of the terms in *patt* is a variable x, and let us suppose that the Default Rule is applied, expanding with the column where this variable appears. The expression *exp*, after a renaming of x given exp', will be in a branch corresponding to each constructor (or **otherwise**). Also, x is replaced in *patt* by n new variables $x_1, \ldots, x_n$, where n is the arity of the constructor. But none of the new variables appears in exp', and that means that any substitution over them does not modify exp'. Only the changes over the rest of the pattern may modify it,

but always in the same form. Thus, all the leaves in the **case** tree which have as associated expression *exp*, after doing all the necessary substitutions, will have the same expression and the sharing will be total.

But it still remains the problem that the number of new variables x_i depends on the constructor in which *exp* is placed, and thus there are an amount of useless bindings for *exp* which depends on the *branch* where the expression *exp* was placed. This can be solved by adding a small code that reorganizes the bindings, or by compiling the expression in such a way that it can deal with these problems. Anyway, it depends on the model of machine in which the compilation is done. Also there exists the same problem for the code duplication for side conditions, but usually these codes are very small, and duplication can be better than sharing.

Related Work

In the framework of functional programming, the first proposed algorithms are [Aug85] and [Wad87]. They give a set of rules to compile functions defined by pattern. Some of our rules in section 3 are similar to those in [Wad87]. The difference is that we have a *default rule* while Wadler's algorithm has several rules to distinguish different cases. The main advantage of our algorithm (out of laziness) with respect to [Aug85] and [Wad87] is that ours has not backtracking in the compiled code (and thus each argument is parsed at most once). Wadler's algorithm has been widely used (for instance in Gofer).

[Grä91] presents a theoretical characterization of an algorithm. The generated code is equivalent to ours if we would always expand with the first column in the *default rule* (i.e. without taking care about laziness).

[Sch88] gives an algorithm that works with restrictions in the types. That means that it can compile functions defined over subtypes of a given type.

The algorithm proposed in [PB85] performs the match *bottom-up*. Our algorithm cannot work in this way, since a *bottom-up* strategy requires the whole evaluation of the expression that it is been matched, and this is against our objective of keeping laziness. Because the match is performed *bottom-up*, it can not work with infinite objects.

In [Lav87] a characterization to know whether a list of patterns is lazy or not is given, but it is very complex and it has a difficult implementation. [Lav88] presents an algorithm (based on [Lav87]), that simplifies that characterization, but it still needs to evaluate if a list of patterns is lazy. With our algorithm it is not necessary to do this characterization previously (which can be very hard to do) because it uses a set of complete rules which will find a lazy strategy (if there exists one). Even if there is no lazy strategy, the compiled code will be *better* than the one generated by the algorithms in [Grä91] or [Sch88], because we can use the local laziness of some subpatterns.

[SRR92] shows that there exist patterns which produce an exponential (in size) tree for any possible strategy. In a recent work, [Mar94] studies lazy algorithms but using backtracking. This approach have some problems, because there exist patterns such that the match leads to a sequential checking.

6 Conclusion

In this paper we have presented an effective algorithm for compiling pattern matching in functional programming languages. This algorithm has a complete set of rules to obtain a code that has no backtracking and that explores the arguments as good as possible in order to preserve laziness. These rules are easy to implement and allow a great variety of adjustment which can improve the generated code. We think that this work method is suitable for this kind of problems and it allows to refine the quality of an algorithm to contain new characteristics. In fact, our intention is to extend the algorithm so it can compile functions that are applied on a subset of its type (as it is done in [Sch88]).

Another advantage of working with a sequence of rules, opposite to give characterizations (as in [Lav88] and [Grä91]), is that in spite of the fact that the problem does not fulfill the property we are characterizing, the rules can be applied to pieces of it. Moreover, it is done without searching these pieces but applying the *best* possible rule in any moment. For instance, there are many patterns which are not lazy because of a small part of the parameters; i.e. there are certain arguments that have to be evaluated following a fixed ordering and there are others for whom this ordering is not necessary. An algorithm of characterization will discard these patterns whereas an algorithm of rules will be able to find most of this arrangement.

Acknowledgements

We would like to thank A. Gavilanes for his help in the first steps of this research.

References

[Aug85] L. Augutsson. Compiling pattern matching. *Functional Programming Languages and Computer Architecture'85, LNCS 201*, pages 368 – 381, 1985.

[Grä91] A. Gräf. Left-to-Right tree pattern matching. *Rewriting Techniques and Applications'91, LNCS 488*, pages 323 – 334, 1991.

[HO82] C.M. Hoffmann and M.J. O'Donell. Pattern matching in trees. *Journal of the ACM*, 29(1):68 – 95, 1982.

[Lav87] A. Laville. Lazy pattern matching in the ML language. *FST & TCS'87, LNCS 287*, pages 400 – 419, 1987.

[Lav88] A. Laville. Implementation of lazy pattern matching algorithms. *ESOP' 88, LNCS 300*, pages 298 – 316, 1988.

[Mar94] Luc Maranget. Two techniques for compiling lazy pattern matching. Technical Report RR–2385, INRIA, 1994.

[PB85] P. W. Purdom and C.A. Brown. Fast many-to-one matching algorithms. *Rewriting Techniques and Applications'85, LNCS 202*, pages 407 – 416, 1985.

[Sch88] Ph. Schnoebelen. Refined compilation of pattern-matching for functional languages. *Science of Computer Programming*, 11:133 – 159, 1988.

[SRR92] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *ICALP'92, LNCS 623*, 1992.

[Wad87] P. Wadler. Efficient compilation of pattern-matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5. Prentice Hall International, 1987.