Adaptive Pattern Matching on Binary Data*

Per Gustafsson and Konstantinos Sagonas

Department of Information Technology, Uppsala University, Sweden
{pergu,kostis}@it.uu.se

Abstract. Pattern matching is an important operation in functional programs. So far, pattern matching has been investigated in the context of structured terms. This paper presents an approach to extend pattern matching to terms without (much of a) structure such as binaries which is the kind of data format that network applications typically manipulate. After introducing a notation for matching binary data against patterns, we present an algorithm that constructs a tree automaton from a set of binary patterns. We then show how the pattern matching can be made adaptive, how redundant tests can be avoided, and how we can further reduce the size of the resulting automaton by taking interferences between patterns into account. The effectiveness of our techniques is evaluated using implementations of network protocols taken from actual telecom applications.

1 Introduction

Binary data are omnipresent in telecom and computer network applications. Many formats for data exchange between nodes in distributed computer systems (MPEG, ELF, PGP keys, yEnc, JPEG, MP3, GIF, ...) as well as most network protocols use binary representations. The main reason for using binaries is size; a binary is a much more compact format than the symbolic or textual representation of the same information. Consequently, less resources are required to transmit binaries over the network.

When binaries are received, they typically need to be processed. Their processing can either be performed in a low-level language such as C (which can directly manipulate these objects), or binaries need to be converted to some non-binary term representation and manipulated by a high-level language such as a functional programming language. The main problem with the second approach is that most high-level languages do not provide adequate support for binary data. As a result, programming tends to become pretty low level anyway (e.g., use of bit-shifts) and the necessary conversion takes time and results in a format which requires more space to be stored. So, both for convenience and out of performance considerations, it is frequent that the first approach is followed; despite the fact that this practice is possibly error-prone and a security risk.

Our aim is to make programming of telecom and packet filter applications using high-level languages both easier and more efficient than its counterpart in low-level languages such as C. More specifically, given a functional programming language which has been enriched with a binary data type and a convenient notation to perform pattern matching on binaries, we propose methods to extend a key feature of functional programs, pattern matching, to binary terms.

* Research supported in part by grants from ASTEC, Vetenskapsrådet, Ericsson, and T-Mobile.

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 124–139, 2004.

Doing so is not straightforward for the following two reasons. First, unlike pattern matching on structured terms where arities and argument positions of constructors are statically known, binary pattern matching needs to deal with the fact that binaries have a totally amorphous structure. Second, typical uses of binaries (e.g. in network protocols) are such that certain parts of the binary (typically its headers) encode information about how many parts the remaining binary contains and how these parts are to be interpreted. An effective binary pattern matching scheme has to effectively cater for these uses.

On the other hand, the potential performance advantages of our approach should be clear: indeed, hand-coded pattern matchers, even in C, can hardly compete with those derived automatically using a systematic algorithm, once the sizes of the patterns becomes significant.

This paper presents an adaptive binary pattern matching scheme, based on *decision trees*, that is tailored to the characteristics of binaries in typical applications. The reason we use decision trees is that they result in fast execution (since each constraint on the matching is tested at most once).[1] Our implementation is based on the Erlang/OTP system; a system which is used to develop large telecom applications where binary pattern matching allows implementation of network protocols using high-level specifications.

The structure of the rest of the paper is as follows: the next section overviews a notation for matching binary data against patterns. Although the notation is that of ERLANG, the ideas behind it are generic. After introducing a definition of what binary pattern matching is (Sect. 3), we present an algorithm that constructs a tree automaton from a set of binary patterns (Sect. 4). We then show how to perform effective pruning (Sect. 4.2), how pattern matching can be made adaptive (Sect. 4.3), how redundant tests can be avoided, and how the size of the resulting automaton can be further reduced by taking interferences between patterns into account (Sect. 5). After reviewing related work (Sect. 6), we evaluate the effectiveness of our techniques (Sect. 7), and conclude.

2 Binaries in Erlang

ERLANG's bit syntax allows the user to conveniently construct binaries and match these against binary patterns. A simplified introduction to ERLANG's bit syntax appears below; for more information, see [12,17].

In ERLANG, a binary is written with the syntax `<<`$\texttt{Seg}_1$`,`$\texttt{Seg}_2$`,...,`$\texttt{Seg}_n$`>>` and represents a sequence of machine bits that are byte-aligned. Each of the $\texttt{Seg}_i$'s specifies a *segment* of the binary, which represents an arbitrary number of contiguous bits in the binary. Segments are placed next to each other in the same order as they appear in the bit syntax expression.

Each segment expression has the general syntax `Value:Size/Specifiers` where both the `Size` and the `Specifiers` fields can be omitted since there are default values for them; see [12]. The `Value` field must however always be specified. In a binary matching expression, the `Value` can either be an Erlang term (of any type) or an unbound variable. The `Size` field, which denotes the number of bits of the segment, can either be statically

[1] However, since the size of the tree automaton is exponential in the worst case, the full version of this paper [8] also presents an alternative approach to compiling binary pattern matching which is conservative, and more specifically linear, in space.

an integer or a variable that will be bound to an integer. The `Specifiers` can be used to specify how the segment should be interpreted (e.g., what is its type). For simplicity of presentation, only `integer` and `binary` specifiers will be used in this paper; see [12] for the complete set of specifiers. Moreover, a `binary` type specifier will mean that this segment is viewed as the *tail* (i.e., the complete remaining part) of the binary. Binary tail segments will have no explicit size specified for them.

Binary Matching. The syntax for matching with a binary if `Binary` is a variable bound to a binary is `<<`$\texttt{Seg}_1$`,`$\texttt{Seg}_2$`,...,`$\texttt{Seg}_n$`>> = Binary`. The $\texttt{Value}_i$ fields of the $\texttt{Seg}_i$ expressions, which describe each segment, will be matched to corresponding segments in `Binary`. For example, if the $\texttt{Value}_1$ field in $\texttt{Seg}_1$ contains an unbound variable and the size of this segment is 16, this variable will be bound to the first 16 bits of `Binary`.

Example 1. As shown below, binaries are often constructed as a sequence of comma-separated unsigned 8 bit integers inside `<<>>`'s, but pattern matching can select parts of binaries whose size is less than a byte long (and actually there are no byte-alignment restrictions). The ERLANG code:

```
<<A1:3/integer, A2:5/integer, B/binary>> = <<42,43,44>>
```

results in the binding `A1=1`, `A2=10`, `B=<<43,44>>`. Here `A1` matches the first three bits. These bits are interpreted as an unsigned, big-endian integer. Similarly, `A2` matches the next five bits. `B` is matched to the rest of the bits of the binary `<<42,43,44>>`. These bits are interpreted as a binary since the segment has this type specifier.

Example 2. `Size` fields of segments are not always statically known. This occurs quite often and complicates the pattern matching operation in our context. The code:

```
<<Sz:8/integer, Vsn:Sz/integer, Msg/binary>> = <<16,2,154,11,12>>
```

results in the binding `Sz=16`, `Vsn=666`, and `Msg=<<11,12>>`.

Example 3. Naturally, pattern matching against a binary can occur in a function head or in a `case` statement just like any other matching operation as in the code below:

```
case Binary of
  <<42:8/integer, X1/binary>> -> X = X1, handle1(X);
  <<Sz:8, V:Sz/integer, X2/binary>> when Sz > 16 -> X = X2, handle2(V,X);
  <<_:8, X3:16/integer, Y:8/integer>> -> X = X3, handle3(X,Y)
end.
```

Here `Binary` will match the pattern in the first branch of the `case` statement if its first 8 bits represented as an unsigned integer have the value 42. In this case, `X1` (and later `X`) will be bound to a binary consisting of the rest of the bits of `Binary`. If this is not the case, then `Binary` will match the second pattern if the first 8 bits of `Binary` interpreted as an unsigned integer are greater than 16. Notice that this is a non-linear and guarded binary pattern. Finally, if `Binary` is exactly 32 bits long, `X` (through `X3`) will be bound to an integer consisting of the second and third byte of the `Binary` (taken in big-endian order). If neither of the patterns match, the whole match expression will fail. On the right, we show three examples of matchings and a failure to match using this code.

`Binary`	matching of `X`
`<<42,14,15>>`	`<<14,15>>`
`<<24,1,2,3,10,20>>`	`<<10,20>>`
`<<12,1,2,20>>`	`258`
`<<0,255>>`	*failure*

3 Binary Pattern Matching Definitions

We assume the usual definition of when two non-binary terms (integers, compound terms, ...) match. A binary pattern matching is defined by a binary term and a set of binary patterns, which is ordered with respect to the priority of the patterns.

In a binary pattern matching compiler, each binary pattern b_i consists of a list of segments $[seg_1, \ldots, seg_n]$ and is associated with a success label which is denoted by $SL(b_i)$.[2] Each segment is represented by a tuple $seg_i = \langle v_i, t_i, p_i, s_i \rangle, i \in \{1, \ldots, n\}$ consisting of a value, a type, a position, and a size field. The value and type fields contain the term in the `Value` field and the type `Specifier` of the corresponding segment, respectively. The size field s_i represents the `Size` of the segment in bits. When the size is statically known, s_i is an integer constant. Otherwise, s_i is either a variable which will be bound to an integer at runtime, or the special don't care variable (denoted as _) which is used when the last segment, seg_n, is of `binary` type without any constraint on its size (cf. the first two binary patterns of Ex. 3). The p_i field denotes the position where segment seg_i starts in the binary. If the size values of all preceding segments are statically known, then p_i is just an integer constant and is defined as $p_i = \sum_{j=1}^{i-1} s_j$. However, the presence of variable-sized segments complicates the calculation of a segment's position. In such cases, we will denote p_i's as $c + V$ where c is the sum of all sizes of preceding segments which are statically known and V is a symbolic representation of the sum of all s_j terms of preceding segments whose values are variables. For example, the binary pattern of Ex. 2 is represented as $[\langle \texttt{Sz}, \texttt{integer}, 0, 8 \rangle, \langle \texttt{Vsn}, \texttt{integer}, 8, \texttt{Sz} \rangle, \langle \texttt{Msg}, \texttt{binary}, 8+\texttt{Sz}, \_ \rangle]$.

Each binary pattern corresponds to a sequence of *actions* obtained by concatenating the actions of its segments. The actions of each segment generally consist of a size and a match test. Each match test includes an associated read action which is to be performed before the actual match test. These are defined below.

Definition 1 (Size test). *For each segment $seg_i = \langle v_i, t_i, p_i, s_i \rangle$ of a binary pattern $[seg_1, \ldots, seg_n]$, if $s_i \neq \_$ we associate a size test st defined as*

$$st = \begin{cases} \texttt{size}(=, p_i + s_i) & \text{if } i = n \\ \texttt{size}(\geq, p_i + s_i) & \text{otherwise} \end{cases}$$

Given a binary b, st succeeds if the size of b in bits is equal to (resp. at least) $p_i + s_i$.

Note that no size test is associated with a tail binary segment (a segment where $s_i = \_$). Also, although positions and sizes might not be integer constants statically, in type-correct programs, they will be so at runtime. Thus the second argument of a size test will always be (evaluable to) an integer constant at runtime.

Definition 2 (Read action). *For a segment $\langle v, t, p, s \rangle$, the corresponding read action (denoted $\texttt{read}(p, s, t)$) is as follows: given a binary b, the action reads s bits starting at position p of b, constructs a term of type t out of them, and returns the constructed term. When $s = \_$ the action reads the remaining part of the binary.*

[2] The fail labels are implicitly defined by the order of the patterns; the fail labels of every $b_k, 1 \leq k \leq n-1$ point to b_{k+1} except for the fail label of b_n which points to a *failure* action.

Definition 3 (Match test). *For a segment $\langle v, t, p, s\rangle$, a match test (which is denoted* `match`(v, ra) *where ra is the corresponding read action) succeeds if the term r returned by ra matches v. If the match test is successful, the variables of v get bound to the corresponding subterms of r.*

Example 4. Action sequences for the three binary patterns in the case statement of Ex. 3 are shown below:

```
b1 = {size(≥,8), match(42,read(0,8,integer)), match(X1,read(8,_,binary))}
b2 = {size(≥,8), match(Sz,read(0,8,integer)),
      size(≥,8+Sz), match(V,read(8,Sz,integer)), match(X2,read(8+Sz,_,binary))}
b3 = {size(=,32), match(X3,read(8,24,integer)), match(Y,read(24,32,integer))}
```

Note that the above action sequences are optimized. Size tests which are implied by other ones have been removed and match tests which do not influence the binary matching have also been eliminated. For example, without any optimizations, b_3 is:

```
b3 = {size(≥, 8), match(_,read(0,8,integer)),
      size(≥,24), match(X3,read(8,24,integer)),
      size(=,32), match(Y,read(24,32,integer))}
```

Since there is a tight correspondence between segments and action sequences, representing a binary pattern using segments is equivalent to representing it using actions. Since actions are what is guiding the binary pattern matching compilation, we henceforth represent binary patterns using action sequences and use the terms binary patterns and action sequences to mean the same thing.

Definition 4 (Static size equality). *Two sizes s_1 and s_2 are statically equal (denoted $s_1 = s_2$) if they are either the same integer or the same variable.*

Definition 5 (Static position equality). *Two positions p_1 and p_2 are statically equal (denoted $p_1 = p_2$) if their representations are identical (i.e., if they are either the same constant, or they are of the form $c_1 + V_1$ and $c_2 + V_2$ where $c_1 = c_2$ and V_1 is the same multiset of variables as V_2).*

Definition 6 (Statically equal read actions). *Two read actions $ra_1 =$* `read`(p_1, s_1, t_1) *and $ra_2 =$* `read`(p_2, s_2, t_1) *are statically equal (denoted $ra_1 = ra_2$) if $s_1 = s_2$, $p_1 = p_2$, and $t_1 = t_2$.*

Definition 7 (Size test compatibility). *Let $|b|$ denote the size of a binary b. A size test $st =$* `size`$(op, p+s)$ *where $op \in \{=, \geq\}$ is compatible with a binary b (denoted $st \sqsubseteq b$) if $(p + s)$ op $|b|$. If the condition does not hold, we say that the size test is incompatible with the binary ($st \not\sqsubseteq b$).*

Definition 8 (Match test compatibility). *Let $ra =$* `read`(p, s, t) *be a read action. A match test $mt =$* `match`(v, ra) *is compatible with a binary b (denoted $mt \sqsubseteq b$) if ra (or more generally a read action which is statically equal to ra) has been performed and the sub-binary of size s starting at position p of b when read as a term of type t matches with the term v. If the term v does not match, we say that the match test is incompatible with the binary ($mt \not\sqsubseteq b$).*

We can now formally define what binary pattern matching is. In the following definitions, let B denote a set of binary patterns ordered by their textual priority.

Definition 9 (Instance of binary pattern). *A binary b is an instance of a binary pattern $b_i \in B$ if b is compatible with all the tests of b_i.*

Definition 10 (Binary pattern matching). *A binary pattern $b_i \in B$ matches a binary b if b is an instance of b_i and b is not an instance of any $b_j \in B, j < i$.*

4 From a Set of Binary Patterns to a Tree Automaton

The construction of the tree automaton begins with an ordered set of k binary patterns which have been transformed to the corresponding action sequences $B = \{b_1, \ldots, b_k\}$. The construction algorithm, shown below, builds the tree automaton for B and returns its start node. Each node of the automaton consists of an action and two branches (a success and a failure branch) to its children nodes. In interior nodes, the action is a test. In leaf nodes, the action is a goto a success label, or a *failure* action.

Given an action a and a set of action sequences B, the action implicitly creates two sets, B_s and B_f. Action sequences in B_s are sequences from B that either do not contain a, or sequences which are created by removing a from them. B_f are action sequences from B that do not contain a. These two sets are driving the construction of the tree automaton. More specifically, the success and failure branches of an interior node point to subtrees that are created by calling the construction algorithm with B_s and B_f, respectively.

The tree automaton operates on an incoming binary b. The algorithm that constructs the tree is quite straightforward. A given node u corresponds to a set of patterns that could still match b when u has been reached. If this set is empty, then no match is possible and a fail leaf is created (lines 2–3). When there are still patterns which can match, the action sequence of the highest priority pattern (b_0) is examined. If it is now empty, then a match has been found (lines 5–7). Otherwise, the `select_action` procedure chooses one of the remaining actions a (a size or match test) from an action sequence in B. This is the action associated with the current node. Based on a, procedures `prune_compatible` and `prune_incompatible` construct the B_s and B_f sets described above. The success and failure branches of the node are then obtained by recursively calling the construction algorithm with B_s and B_f, respectively (lines 9–14).

```
Procedure Build(B)
 1. u := new_node()
 2. if B = ∅ then
 3.   u.action := failure
 4. else
 5.   b_0 := the action sequence of the
             highest priority pattern in B
 6.   if current_actions(b_0) = ∅ then
 7.     u.action := goto(SL(b_0))
 8.   else
 9.     a := select_action(B)
10.     u.action := a
11.     B_s := prune_compatible(a, B)
12.     u.success := Build(B_s)
13.     B_f := prune_incompatible(a, B)
14.     u.fail := Build(B_f)
15. return u
```

The `select_action` procedure controls the traversal order of patterns, making the pattern matching adaptive. It is discussed in Sect. 4.3. The `prune_*` procedures can be more effective in the amount of pruning that they perform. This is discussed in Sect. 4.2.

Notice that the match tests naturally handle non-linearity in the binary patterns. Also, although not shown here, it is quite easy to extend this algorithm to allow it to handle guarded binary patterns. The only change that needs to be made is to add guard actions to the action sequences.

4.1 Complexity

The worst case for this algorithm is when no conclusions can be drawn to prune actions and patterns from B. In this case, the size of the constructed tree automaton is exponential in the number of actions (segments) in a pattern. The time complexity for the worst case path through this tree is linear in the total number of segments.

4.2 Pruning

Let a be an action of a node. Based on a, procedure `prune_compatible` creates a pruned set of action sequences by removing a (or more generally actions which are implied by a) and action sequences which contain a test a' that will fail if a succeeds. Similarly, procedure `prune_incompatible` creates a pruned set of action sequences by removing action sequences which contain a test a' that will fail if a fails, and actions that succeed if a fails. The functionality of these procedures can be described as follows:

`prune_compatible`(a, B) Removes all actions from action sequences in B which can be proven to be compatible with any binary b such that $a \sqsubseteq b$ and all action sequences that contain an action which can be proven to be incompatible with any binary b such that $a \sqsubseteq b$.

`prune_incompatible`(a, B) Removes all actions from action sequences in B which can be proven to be compatible with any binary b such that $a \not\sqsubseteq b$ and all action sequences that contain an action which can be proven to be incompatible with any binary b such that $a \not\sqsubseteq b$.

Size test pruning. Using size tests to prune the tree automaton for binary pattern matching is similar to switching on the arity of constructors when performing pattern matching on structured terms. If $=$ were the only comparison operator in size tests, the similarity would be exact. Since in binary pattern matching the size test operator can also be $\geq$ and sizes of segments might not be statically known, the situation in our context is more complicated.

To effectively perform size test pruning we need to setup rules that allow us to infer the compatibility or incompatibility of some size test st_1 with any binary b given that another size test st_2 is either compatible or incompatible with b.

In order to construct these rules we need to describe how size tests can be compared at compile time. Consider a size test, $st = \texttt{size}(op, se)$ where op is a comparison operator and se a size expression. In the general case, the size expression will have the form $c+V$

Table 1. Size pruning rules.

(a) Rules for `prune_compatible`(st, B)

op	op_i	relation	conclusion
$\geq$	$\geq$	$se \geq se_i$	$st_i \sqsubseteq b$
$\geq$	$=$	$se > se_i$	$st_i \not\sqsubseteq b$
$=$	$\geq$	$se \geq se_i$	$st_i \sqsubseteq b$
$=$	$\geq$	$se < se_i$	$st_i \not\sqsubseteq b$
$=$	$=$	$se = se_i$	$st_i \sqsubseteq b$
$=$	$=$	$se \neq se_i$	$st_i \not\sqsubseteq b$

(b) Rules for `prune_incompatible`(st, B)

op	op_i	relation	conclusion
$\geq$	$\geq$	$se_i \geq se$	$st_i \not\sqsubseteq b$
$\geq$	$=$	$se_i \geq se$	$st_i \not\sqsubseteq b$
$=$	$=$	$se = se_i$	$st_i \not\sqsubseteq b$

where c is a constant and V is a multiset of variables. The following definition of how to statically compare size expressions is based on what can be inferred about two different size expressions $c_1 + V_1$ and $c_2 + V_2$, assuming that during run-time, all variables in V_1 and V_2 will be bound to non-negative integers (or else a runtime type exception will occur).

Definition 11 (Comparing size expressions statically). *Let $se_1 = c_1 + V_1$ and $se_2 = c_2 + V_2$ be two size expressions.*

- *se_1 is statically equal to se_2 (denoted $se_1 = se_2$) if $c_1 = c_2$ and V_1 is the same multiset as V_2;*
- *se_1 is statically bigger than se_2 (denoted $se_1 > se_2$) if $c_1 > c_2$ and V_1 is a superset of V_2;*
- *se_1 is statically bigger or equal to se_2 (denoted $se_1 \geq se_2$) if $se_1 > se_2$ or $se_1 = se_2$ or $c_1 = c_2$ and V_1 is a superset of V_2;*
- *se_1 is statically different than se_2 (denoted $se_1 \neq se_2$) if either $se_1 > se_2$ or $se_1 < se_2$.*

Let b be any binary such that a size test $st \sqsubseteq b$ (is compatible with b). In the `prune_compatible`(st, B) procedure we want to prune all size tests st_i such that st_i is contained in some action sequence in B and $st_i \sqsubseteq b$. We also want to prune all action sequences in B that contain a size test st_j such that $st_j \not\sqsubseteq b$. If $st = \texttt{size}(op, se)$ and $st_i = \texttt{size}(op_i, se_i)$ then Table 1(a) presents the conclusion which can be drawn about the compatibility of st_i with b given values for op and op_i and a static comparison of size expressions se and se_i.

Now let b be any binary such that a size test $st \not\sqsubseteq b$ (is incompatible with b). The `prune_incompatible`(st, B) procedure will prune all action sequences in B that contain a size test st_i such that $st_i \not\sqsubseteq b$. Table 1(b) presents when it is possible to infer this size test incompatibility given values for op, op_i, and a static comparison of se and se_i.

The following example illustrates size test pruning.

Example 5. Let $st = \texttt{size}(=, 24 + \texttt{Sz})$ and $B = \{b_1, b_2, b_3, b_4\}$ where:

$$
\begin{aligned}
b_1 &= \{\texttt{size}(=, 24 + \texttt{Sz}), a_{1,2}, \ldots, a_{1,n_1}\} \\
b_2 &= \{\texttt{size}(\geq, 24), a_{2,2}, \ldots, a_{2,n_2}\} \\
b_3 &= \{\texttt{size}(=, 16), \ldots\} \\
b_4 &= \{\texttt{size}(\geq, 32 + \texttt{Sz}), \ldots\}
\end{aligned}
$$

and let $a_{i,j}$ be actions whose size expressions cannot be compared with the size expression of st statically. Then $\texttt{prune\_compatible}(st, B) = \{b'_1, b'_2\}$ where $b'_1 = \{a_{1,2}, \ldots, a_{1,n_1}\}$, and $b'_2 = \{a_{1,2}, \ldots, a_{2,n_2}\}$. Why the size test st is removed from b_1 should be obvious. In b_2, the size test $\texttt{size}(\geq, 24)$ is implied by st (see third row of Table 1(a)) and is removed. Sequences b_3 and b_4 each contain a size test which is false if st succeeds (this is found by looking at rows six and four of Table 1(b)) and are pruned. We also have that $\texttt{prune\_incompatible}(st, B) = \{b_2, b_3, b_4\}$.

Match test pruning. A simple form of match test pruning can be based on the concept of similarity. Let b be a binary and $mt_1 = \texttt{match}(v_1, ra_1)$ and $mt_2 = \texttt{match}(v_2, ra_2)$ be two match tests whose read actions ra_1 and ra_2 are statically equal. If $v_1 = v_2$, then we have the following rules:

$$mt_1 \sqsubseteq b \Rightarrow mt_2 \sqsubseteq b$$
$$mt_1 \not\sqsubseteq b \Rightarrow mt_2 \not\sqsubseteq b$$

If both v_1 and v_2 are constants and $v_1 \neq v_2$, we get the additional rule:

$$mt_1 \sqsubseteq b \Rightarrow mt_2 \not\sqsubseteq b$$

In Sect. 5.3 we describe how to extract more information from the success or failure of a match test by taking interference of actions into account. Doing so, increases the effectiveness of match test pruning.

4.3 Selecting an Action

The `select_action` procedure controls the traversal order of actions and makes the binary pattern matching adaptive. It also allows discussion of the binary matching algorithm without an *a priori* fixed traversal order.

For the binary pattern matching problem, there are constraints on which actions can be selected from the action sequences. A size test can not be chosen unless its size expression can be evaluated to a constant. Similarly, match tests whose read actions have a yet unknown size cannot be selected. More importantly, a match test cannot be selected unless all size tests which precede it have either been selected or pruned. This ensures the safety of performing the read action which a match test contains: otherwise a read action could access memory which lies outside the memory allocated to the binary.

What we are looking for is to select actions which perform effective pruning and thus make the size of the resulting tree automaton small. Since both constructing an optimal decision tree and a minimal order-containing pruned trie are NP-complete problems [9, 5], we employ heuristics. One such heuristic is to select an action which makes the size of the success subtree of a node small. Such actions are defined below.

Definition 12 (Eliminators). *Let $B = \{b_1, \ldots, b_k\}$ be a set of action sequences. A test α (of some $b_j \in B$) is an eliminator of m sequences if exactly m members of B contain a test which will not succeed if α succeeds. A test α is a perfect eliminator if it is an eliminator of $k-1$ sequences. A test α is a good eliminator if it is an eliminator of m sequences and for all $l > m$ there do not exist eliminators of l sequences.*

So, what we are looking for is good eliminators; ideally, for perfect ones. If a perfect eliminator exists each time the `select_action` procedure is called, then the size of the tree automaton will be linear in the total number of actions. Also, the height of the tree (which controls the worst time it takes to find a matching) will be no greater than the number of patterns plus the maximum number of actions in one sequence.

In the absence of perfect eliminators, the heuristics below can be used. Some of them reduce the size of the tree and some the time needed to find a matching.

Eliminator A good eliminator is chosen. As a tie-breaker, a top-down, left-to-right order of selecting good eliminators is followed.

Pruning The action which minimizes the size of the sets of action sequences returned by the `prune_*` procedures is chosen. A top-down, left-to-right order is used as a tie-breaker.

Left-to-Right This is the commonly used heuristic of selecting actions in a top-down, left-to-right fashion. This heuristic does not result in adaptive binary pattern matching, but on the other hand it is typically effective as the traversal order is the one that most programmers would expect (and often program for!); see also [13].

5 Optimizations

The algorithm presented in Sect. 4 is quite inefficient when it comes to the size of the resulting tree automaton. We now present some optimizations that can decrease its size; often very effectively.

5.1 Generating Shorter Action Sequences

Simple ways to avoid unnecessary work are to eliminate size tests which are implied by other ones and to not generate match tests (and their corresponding read actions) for variables that are unused. We have already shown these optimizations in Ex. 4. Another easy way to make the tree more compact is to preprocess the original binary patterns so that they contain fewer segments. Two adjacent segments with constant terms as values can be coalesced into one so that only one match test is generated for them. For example, the pattern `<<0:8, 1:8, B/binary>>` can be source-transformed to the equivalent binary pattern `<<1:16, B/binary>>`.

5.2 Turning the Tree Automaton into a DAG

Creating a directed acyclic graph (DAG) instead of a tree is a standard way to decrease the size of a matching automaton. One possible choice is to construct the tree automaton first, and then use standard FSA minimization techniques to create the optimal DAG. This is however impractical, since it requires that a tree automaton of possibly exponential size is first constructed. Instead, we can use a concept similar to memoization to construct the DAG directly. We simply remember the results we got from calling the `Build` procedure, and if the procedure is called again with the same input argument we return the subtree that was constructed at that time.

Turning a tree into a DAG does not affect the time it takes to perform binary pattern matching. This is evident since the length of paths from the root to each leaf is not changed. It is difficult to formalize the size reduction obtained by this optimization, as it depends on the characteristics of the action sequences and its interaction with action pruning. In general, the more the pruning that the selected actions perform, the harder it is to share subtrees. However, in our experience, turning the tree into a DAG is an effective size-reducing optimization in practice.

5.3 Pruning Based on Interference of Match Tests

Pruning based on match tests (Sect. 4.2) takes place when two match tests contain read actions which are statically equal. We can increase the amount of pruning performed based on match tests by taking interferences between match tests into account. Consider the following example:

Example 6. In binary patterns $b_1 =$ `<<Sz:4,0:12,X:Sz>>` and $b_2 =$ `<<255:8,...>>` there are not any statically equal read actions in match tests. However, it is clear that if the match test associated with the second segment of b_1 succeeds, then b_2 cannot possibly match the incoming binary. This is because these match tests *interfere*. The notion is formalized below.

Definition 13 (Interference). *Let p_1 and p_2 be statically known positions and $p_1 \leq p_2$. Also, let s_1 and s_2 be statically known sizes. Match tests* $\texttt{match}(v_1, \texttt{read}(p_1, s_1, t_1))$ *and* $\texttt{match}(v_2, \texttt{read}(p_2, s_2, t_2))$ *interfere if $p_1 + s_1 > p_2$. Their common bits are bits in the range $[p_2, \ldots, min(p_2 + s_2, p_1 + s_1)]$.*

For pruning purposes, the concept of interfering match tests is only interesting when both terms v_1, v_2 of the match tests are known. Let us denote the common bits of v_1 and v_2 by v_1' and v_2', respectively.

Definition 14 (Enclosing match test). *Let* $mt_1 = \texttt{match}(v_1, \texttt{read}(p_1, s_1, t_1))$ *and* $mt_2 = \texttt{match}(v_2, \texttt{read}(p_2, s_2, t_2))$ *be two match tests which interfere and let $p_1 \leq p_2$. We say that mt_1 encloses mt_2 (denoted $mt_1 \supseteq mt_2$) if $p_1 + s_1 \geq p_2 + s_2$.*

Let mt_1 and mt_2 be match tests which interfere and v_1' and v_2' be their common bits. Then mt_2 will be:

1. compatible with all binaries that mt_1 is compatible with if $v_1' = v_2'$ and $mt_1 \supseteq mt_2$;
2. incompatible with all binaries that mt_1 is compatible with if $v_1' \neq v_2'$;
3. incompatible with all binaries that mt_1 is incompatible with if $mt_2 \supseteq mt_1$ and $v_1' = v_2'$.

The first two rules can be used in the `prune_compatible(`mt_1, B`)` procedure to prune interfering match tests. Similarly, the last rule can be used to guide the pruning in the `prune_incompatible(`mt_1, B`)` procedure.

5.4 Factoring Read Actions

To ease exposition of the main ideas, we have thus far presented read actions as tightly coupled with match actions although they need not really be. Indeed, read actions can appear in the action field of tree nodes. Such tree nodes need a success branch only (their failure branch is null). With this as the only change, read actions can also be selected by the `select_action` procedure, statically equal read actions can be factored, and read actions can be moved around in the tree (provided of course that they are still protected by the size test that makes them safe).

Since, especially in native code compilers, accessing memory is quite expensive, one important optimization is to avoid unnecessary read actions. This can be done if there is a read action that is statically equal to a read action which has already been performed. Then the result of the first read action can be saved in some temporary register, and the second read action can be replaced by a use of that register. Our experience is that in practice this optimization significantly reduces the time to perform binary pattern matching.

Also, to reduce code size, standard compiler techniques like *code hoisting* can be used to move a read action to a node in the tree where a statically equal read action will be performed on all successful paths from that node to a leaf. These read actions can then be removed, reducing the code size.

6 Related Work

In functional languages, compilation schemes for efficient pattern matching over structured terms have been developed and deployed for more than twenty years. Their main goal has been to make the right trade-off between time and space costs. The *backtracking automaton* approach [1,15] is *a priori* economical in space usage (because patterns never get compiled more than once) but is inefficient in time (since the same symbols can be inspected several times). This is the approach used in implementations of typed functional languages such as Caml and Haskell. In the context of the Objective-Caml compiler, [10] suggested using exhaustiveness and incompatibility characteristics of patterns to improve the time behavior of backtracking automata. Exhaustiveness is only applicable when constructor-based type definitions are available, and thus cannot be used in binary pattern matching. In our context, a kind of incompatibility-based pruning is obtained by taking advantage of match test interference (Sect. 5.3).

Deterministic tree automata approaches have been proposed before; e.g. [3,14]. Such tree-based approaches guarantee that no constructor symbol is inspected twice, but doing so leads to exponential upper bounds on the automaton size. One way of dealing with this problem is to try to construct an optimal traversal order to minimize the size of the tree. However, since the optimization problem is NP-complete, [3] argues that heuristics should be employed to find near-optimal trees. In the same spirit, [14] also suggests several different heuristics to synthesize an adaptive traversal order that results in a tree automaton of small size. To further decrease the size of the automaton they generate a directed acyclic graph (DAG) automaton by sharing all isomorphic subtrees. Finally, [13] also examines several different match-compilation heuristics (including those of [3,14])

and measures their effects on different benchmarks. However, all these works heavily rely on being able to do a constructor-based decomposition of patterns, and to inspect terms in positions which are known statically.

By exploiting the foreign language interface, an API for a bit stream data structure for Haskell is introduced in [16]. Pattern matching on these bit streams is however not explored. Some of the techniques presented here could likely be used to implement pattern matching on bit streams for Haskell which would allow for a less imperative style of programming. There are however some fundamental differences between our work and [16] as the lazy setting of [16] might restrict the traversal order of tests.

Several packet filtering frameworks have been developed by the networking community. Some of them, e.g., PathFinder [2], DPF [6] and BPF+ [4], use the backtracking automaton approach to pattern matching to filter packets. To achieve better performance common prefixes are collapsed in [2,6]. In contrast, [4] employs low level optimizations such as redundant predicate elimination to produce efficient pattern matching code. As far as we know, the tree automaton approach to pattern matching compilation has not been used in packet filters. We intend to investigate the effectiveness of our scheme in a packet filter setting. Finally, [11] proposes an external type system for packet data which allows for type checking of packets and suggests a scheme to use pattern matching based on type refinement to construct efficient packet filters.

7 Some Experiments

In [7], we presented a scheme for efficient just-in-time compilation of BEAM instructions that manipulate binaries to native code.[3] On a set of benchmarks, when executing native code but without pattern matching compilation, speedups ranging from 20% to four times faster compared with BEAM were obtained. With [7] providing an efficient basis for compiling binary instructions to native code, and with adaptive pattern matching complementing nicely that work, we felt there is no need to do extensive benchmarking in this paper. We just report on two issues.

7.1 Impact of Pruning Heuristics and Optimizations

As benchmark programs we selected three different (parts of) actual protocol applications that perform binary pattern matching. The `BER-decode` matching code is quite complicated; it contains 14 different patterns and 10 distinct read actions. `BS-extract` contains just 4 patterns and 11 distinct read actions (each pattern contains a perfect eliminator; adaptive selection is required to benefit from it). The `PPP-config` matching code contains 8 different patterns and 7 distinct read actions. Using these benchmarks, we measured the impact of different heuristics used in the `select_action` function. The Eliminator, Pruning, and Left-to-Right heuristics are as described in Sect. 4.3. Both size (the number of nodes in the resulting DAG) and two time-related aspects of the heuristics are reported: the average (resp. maximum) height of the DAG measured as the average length of paths (resp. longest path) from a start node to a leaf node.

[3] BEAM is the virtual machine of the Erlang/OTP (Open Telecom Platform) system. Native code compilation of binaries is available in Erlang/OTP since Oct. 2002; see www.erlang.org.

Table 2. Impact of heuristics and optimizations.

	BER-decode			BS-extract			PPP-config		
Heuristic	Size	Avg. H	Max H	Size	Avg. H	Max H	Size	Avg. H	Max H
Eliminator	101	15.30	17	28	17.5	19	40	8.73	10
Pruning	74	14.31	17	28	17.5	19	41	8.73	10
Left-to-Right	78	14.36	17	43	17.5	19	46	10.93	16
Read Hoisting	66	15.50	17	22	17.5	19	28	10.90	15

Table 3. Comparison between programs manipulating binary data written in C and in ERLANG.

Program written in	Time
C returning its result as a binary	2220
ERLANG using binary pattern matching	2580
C returning its result as an Erlang term	4110
ERLANG processing the data in the binary represented using a list of integers	41060

In Table 2, the Read Hoisting row refers to an optimization which aggressively uses code hoisting to move read actions up to a node if statically equal read actions exist on at least two paths from that node. Therefore this optimization yields tree automata that are small in size. However, the time properties of these automata are rarely better and actually sometimes worse than those for automata created using the Left-to-Right heuristic. The Eliminator and Pruning heuristics give similar time characteristics for these benchmarks, but it seems that the Pruning heuristic yields automata which are both small in size and with better matching times. As optimizing for time is our current priority, we find the Pruning heuristic to be the most suitable choice. We are currently using it as default.

7.2 Speed of Binary Pattern Matching in Erlang

Speed is critical in programs implementing telecom and network protocols. It is quite common for developers to resort to low-level languages such as C in order to speed up the time-critical parts of their applications, and indeed manipulating bit sequences is considered C's bread and butter. So, we were curious to know how well binary pattern matching in ERLANG compares with manipulating binaries in C.

We found four different versions of the same program whose input is a binary. The benchmark is taken from the ASN.1 library available in the Erlang/OTP distribution. Two versions written in C exist: one which is a stand alone program (first row of Table 3) and one which is used as a linked in C-driver in an application which is otherwise written in ERLANG. The latter thus needs to return its output in the form of an ERLANG term, and a translation step takes place as the last step of the C program. The other two versions are written completely in ERLANG: one manipulates its input as a binary, performs binary pattern matching and returns a result as an ERLANG term for further processing, while the last version receives its input as a list of integers (a representation which could be a reasonable choice if binaries were not part of the language).

As seen in Table 3, showing times in msecs, the stand-alone C program (compiled using `gcc -O3`) is the fastest program but is only about 15% faster than the ERLANG code using adaptive binary pattern matching. When the rest of the application is written in ERLANG, and a translation step in needed for the C program to be used as a linked-in driver, the ERLANG code with binary pattern matching is about 60% faster. Using a list of integers representation rather than a binary data type results in a program with a rather poor performance. It should be mentioned that the ERLANG programs have been run with a rather large heap to avoid garbage collections (which C does not perform).

8 Concluding Remarks

From the performance data in Table 3, it should be clear that enriching a functional programming language with a binary data type and implementing a binary pattern matching compilation scheme such as the ones described in this paper are additions to the language which are worth their while. Indeed, since 2000 when a notation for binary pattern matching was introduced to ERLANG [12], binaries have been heavily used in commercial applications and programmers have often found innovative uses for them.

Our adaptive binary pattern matching compilation scheme will soon find its way into Erlang/OTP R10 (release 10) from Ericsson, and ERLANG programmers will no doubt benefit from it. However, the ideas we presented are generic and as we have shown there is nothing that prevents binaries from being first-class citizens in declarative languages. For this reason, we hope that other high-level programming languages, which employ pattern matching, will also benefit from them.

References

1. L. Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, number 201 in LNCS, pages 368–381. Springer-Verlag, Sept. 1985.
2. M. Bailey, B. Gopal, M. Pagels, L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of USENIX OSDI Symposium*, pages 115–123, Nov. 1994.
3. M. Baudinet and D. MacQueen. Tree pattern matching for ML. Unpublished paper, 1985.
4. A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *ACM SIGCOMM*, pages 123–134, Aug. 1999.
5. D. Comer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24(3):428–440, July 1977.
6. D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM*, pages 53–59. Aug. 1996.
7. P. Gustafsson and K. Sagonas. Native code compilation of Erlang's bit syntax. In *Proceedings of ACM SIGPLAN Erlang Workshop*, pages 6–15. ACM Press, Nov. 2002.
8. P. Gustafsson and K. Sagonas. Adaptive pattern matching on binary data. Technical Report, Department of Information Technology, Uppsala University, Sweden, Dec. 2003.
9. L. Hyafil and R. L. Rivest. Constructing optimal binary decision tress is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.
10. F. Le Fessant and L. Maranget. Optimizing pattern matching. In *Proceedings of the ACM SIGPLAN International Conference on Functional programming*, pages 26–37. Sept. 2001.
11. P. J. McCann and S. Chandra. Packet types: Abstract specification of network protocol messages. In *Proceedings of ACM SIGCOMM*, pages 321–333. Aug./Sept. 2000.

12. P. Nyblom. The bit syntax - the released version. In *Proceedings of the Sixth International Erlang/OTP User Conference*, Oct. 2000. Available at http://www.erlang.se/euc/00/.
13. K. Scott and N. Ramsey. When do match-compilation heuristics matter? Technical Report CS-2000-13, Department of Computer Science, University of Virginia, May 2000.
14. R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM Journal of Computing*, 24(6):1207–1234, Dec. 1995.
15. P. Wadler. Efficient compilation of pattern matching. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 7. Prentice-Hall, 1987.
16. M. Wallace and C. Runciman. The bits between the lambdas: Binary data in a lazy functional language. In *Proceedings of ACM SIGPLAN ISMM*, pages 107–117. ACM Press, Oct. 1998.
17. C. Wikström and T. Rogvall. Protocol programming in Erlang using binaries. In *the Erlang/OTP User Conference*, Oct. 1999. Available at http://www.erlang.se/euc/99/.