JFP **16** (1): 35–74, 2006. © 2005 Cambridge University Press
doi:10.1017/S0956796805005745 Printed in the United Kingdom

*Efficient manipulation of binary data using pattern matching**

PER GUSTAFSSON and KONSTANTINOS SAGONAS
Department of Information Technology, Uppsala University, Sweden
(*e-mail:* {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 article 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 the binary datatype and a notation for matching binary data against patterns, we present an algorithm that constructs a decision tree automaton from a set of binary patterns. We then show how the pattern matching using this tree automaton 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. Since the size of the tree automaton is exponential in the worst case, we also present an alternative new approach to compiling binary pattern matching which is conservative in space and analyze its complexity properties. The effectiveness of our techniques is evaluated using standard packet filter benchmarks and on implementations of network protocols taken from actual telecom applications.

1 Introduction

Binary data are omnipresent in telecommunication and computer network applications. Many formats for data exchange between nodes in distributed computer systems (MPEG, ELF, PGP keys, yEnc, JPEG, MP3, GIF, etc.) 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. As a consequence, less resources are required for binaries to be transmitted 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 they need to be converted to some term representation and then manipulated in 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 common operations on binary data.

* Research supported in part by grant #621-2003-3442 from the Swedish Research Council (Vetenskapsrådet) and by the Vinnova ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson AB and T-Mobile, U.K.

As a result, programming tends to become pretty low level anyway, e.g. littered with bit-shifting operations. Also, the necessary conversion to a structured term representation takes time and results in a format which requires more storage space. So, both for convenience and out of performance considerations, it is more often than not the case that the low-level approach is followed; despite the fact that this practice is possibly error-prone and opens up possibilities for security holes.

Our aim is to make programming of telecom and packet filter applications using high-level languages both convenient and natural and at the same time as efficient as 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.

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, i.e. the patterns often contain repeated occurrences of variables not all of which can be translated away using explicit equality tests. An effective binary pattern matching scheme has to cater for these uses.

On the other hand, the potential performance advantages of our aim should be clear, at least to functional programmers. Indeed once the number and sizes of the patterns become significant, hand-coded pattern matchers, even when written in low-level languages such as C, can hardly compete with those derived automatically using systematic algorithms like those presented in this article.

The main part of this article 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 faster code (since each constraint on the matching is tested at most once), and fast execution is one of the main goals in our application domain. However, since the size of the decision tree can be in the worst case exponential in the number of patterns, we also present an alternative approach whose worst case space requirement is linear in the total number of matching tests. Our implementation vehicle is Erlang/OTP (Open Telecom Platform), 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 this article is as follows: the next section overviews a notation for creating binaries and for matching binary data against patterns. Although the notation and syntax is that which is used in the ERLANG language, the ideas behind it are generic. Examples of how binary pattern matching can be used for common programming tasks in network protocol processing are presented in section 3. After introducing a definition of binary pattern matching in section 4, we present an algorithm that constructs a decision tree automaton from a set of binary patterns (section 5). In particular, we show how to perform effective pruning, how pattern matching can be made adaptive, 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. An approach to compiling binary pattern matching which is conservative in space, and its complexity properties, are presented in section 6. After evaluating the effectiveness of our techniques on packet filter benchmarks and on implementations of network protocols from actual telecom applications (section 7), we review related work (section 8), and conclude in section 9.

2 Binaries

The binary datatype represents a finite sequence of 8-bit bytes. Two basic operations can be performed on a binary: *creation* of a new binary and *matching* against an existing binary.

2.1 Creation of binaries using the bit syntax

ERLANG's bit syntax (described in Nyblom (2000) but see also Wikström & Rogvall (1999)) allows the user to conveniently construct binaries and match these against binary patterns. A bit syntax expression (called a Bin in Nyblom (2000)) is the building block used to both construct binaries and match against binary patterns. A Bin is written with the following syntax:

<<Segment$_1$, Segment$_2$, ..., Segment$_n$>>

The Bin represents a sequence of bytes. Each of the Segment$_i$'s specifies a *segment* of the binary. A segment represents an arbitrary number of contiguous bits in the Bin. The segments are placed next to each other in the same order as they appear in the bit syntax expression.

2.1.1 Segments

Each segment expression has the general syntax:

```
Value:Size/SpecifierList
```

where both the `Size` and the `SpecifierList` are optional. When they are omitted, default values are used for these specifiers. The `Value` field must however always be specified. In a binary match, the `Value` can either be an Erlang term, a bound variable, an unbound variable, or the don't care variable '_'. The `Size` field can either be an integer constant or a variable that is bound to an integer. The `SpecifierList` is a dash-separated list of up to four specifiers that specify type, signedness, endianess, and unit. The different forms of type specifiers are shown in Table 1 together with a brief description of their use; they are explained in detail below. If all of these type specifiers are used, the syntax of the segment expression is:

```
Value:Size/Type-Signedness-Endianess-unit:Unit
```

The `Size` specifier gives the size of the segment measured in units. Thus the size of the segment in bits (hereafter called its *effective size*) will be `Size * Unit`.

Table 1. *Binary segment specifiers: short description*

`integer`	The segment's bit sequence will be interpreted as an integer. (default)
`float`	The segment's bit sequence will be interpreted as a float. The segment's size can then only be 32 or 64.
`binary`	The segment's bit sequence will not be interpreted. The default `unit` size of a binary is 8.
The following three specifiers apply to integers and floats only.	
`big`	The segment's bytes are in big-endian order. (default)
`little`	The segment's bytes are in little-endian order.
`native`	The segment's bytes are in the byte ordering of the machine on which the program runs.
The following two specifiers apply to integer segments only.	
`signed`	The segment's bit sequence will be interpreted as an integer in 2's complement representation.
`unsigned`	The segment's bit sequence will be interpreted as an unsigned integer. (default)
`unit`	Always followed by '`:`' and an integer between 1 and 256 which denotes the unit size. The unit size is used to determine the segment's *effective size* which is the product of the unit size and the `Size` field. The unit is typically used to ensure either byte-alignment in a binary match or that a new binary has a size that is divisible by 8 regardless of the value of the `Size` field. The default unit size is 1 for integers and floats and 8 for binaries.

2.1.2 Types

The bit syntax allows three different types to be specified for segments of binaries: integers, floats, and binaries:

- The `integer` type specifier is the default and the segment can then be of any size. For integers, the user can also specify endianess and signedness (see Table 1). If unspecified, the default specifiers for an integer segment are a size of 8 bits, unsigned, big-endian, and a unit of 1.
- The `float` type specifier only allows effective sizes of 32 or 64 bits. The user can also specify endianess. The default specifiers for a float segment are a size of 64 bits, a big-endian format, and a unit of 1.
- The `binary` type specifier allows effective sizes that are evenly divisible by 8. Specifying endianess or signedness does not modify how a binary is matched. The default specifiers for a binary segment is the size `all` which means the binary is being matched out completely. If the size of the segment is specified, the default unit used is 8 bits.

Table 2. *Some binary segments and their default expansions*

Segment	Default expansion
`X`	`X:8/integer-unsigned-big-unit:1`
`X/float`	`X:64/float-big-unit:1`
`X/binary`	`X:all/binary`
`X:Size/binary`	`X:Size/binary-unit:8`

2.1.3 Endianess

An endianess specifier determines the order in which bytes form an integer or a float are stored. The specifier `big` means that the bytes are in big-endian order, while the specifier `little` signifies that the bytes are in little-endian order. For example, the bit syntax expression `<<298:16/integer-big>>` is equivalent to `<<1,42>>`, whereas the expression `<<298:16/integer-little>>` is equivalent to `<<42,1>>`.

2.1.4 Signedness

A signedness specifier allows matching of either signed or unsigned integers. The default value is `unsigned`. This means that the segment will be interpreted as an unsigned integer. The `signed` specifier makes sure that the segment is interpreted as an integer in two's complement representation. We note that the `signed` and `unsigned` specifiers are actually allowed in all expressions, but they only have a meaning when used in binary segments whose type is `integer`.

2.1.5 Tail of a binary

As mentioned, if the `binary` type specifier is used without an explicit size specifier, its size gets expanded to the size `all` by default. In the last, when this is not also the first, segment of a binary this use is similar to the familiar list `cdr` operator since a size of `all` means that the binary is matched against the complete remaining binary. (We further discuss the `cdr` similarity in Example 1 in the following section.) A segment of `binary` type however, must be a sequence of 8-bit bytes (i.e. have a size which is evenly divisible by eight). This also applies when a don't care variable is used as `Value`.

2.1.6 Default expansions

All specifiers have default values and sometimes the defaults depend on the values of other specifiers. To summarize the rules which apply, we show how some segments are expanded in Table 2.

2.2 Binary matching

The syntax for matching if `Binary` is a variable bound to a binary is as follows:

$$\texttt{<<Segment}_1\texttt{, Segment}_2\texttt{, ..., Segment}_n\texttt{>> = Binary}$$

The $\texttt{Value}_i$ fields of the $\texttt{Segment}_i$ expressions that describe each segment will be matched to the corresponding segment in `Binary`. For example, if the $\texttt{Value}_1$ field in $\texttt{Segment}_1$ contains an unbound variable and the effective size of this segment is 16, this variable will be bound to the first 16 bits of `Binary`. How these bits will be interpreted is determined by the `SpecifierList` of $\texttt{Segment}_1$.

Example 1
As shown below, binaries are generally written (and also printed) as a sequence of comma-separated unsigned 8 bit integers inside <<>>'s. The ERLANG code:

```
Binary = <<10,11,12>>, <<A:8,B/binary>> = Binary
```

results in the binding `A = 10, B = <<11,12>>`.[1]

Here `A` matches the first 8 bits of `Binary`. Because of the default values (cf. Table 2), these eight bits are interpreted as an unsigned, big-endian integer. `B` is matched to the rest of the bits of `Binary`. These bits are interpreted as a binary since that type specifier has been chosen. Because of that, `B` matches to the complete remaining part of `Binary`, as this is the default size for the `binary` type specifier.

The correspondence of the tail of a binary with the `cdr` operator of lists can be seen by comparing the code given above with the ERLANG code shown below.

```
List = [10,11,12], [A|B] = List
```

The similarity of binaries with lists is even more apparent in Example 7 below.

`Size` fields of segments are not always statically known. This is actually quite a common case and complicates the pattern matching operation in our context. It is also possible that the value of the size field is decided by the matching of a variable in some other, earlier segment. In other words, the patterns are often *non-linear* in the sense that they contain repeated occurrences of variables. This is illustrated with the following example.

Example 2
The ERLANG code:

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

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

Note that in the above binary pattern the repeated occurrences of the `Sz` variable cannot be translated away by using explicit equality tests in the form of guards. For example, the above binary pattern cannot be translated to:

```
<<Sz:8/integer, Vsn:Sz1/integer, Msg/binary>> when Sz == Sz1
```

because this would effectively require a binary pattern matching automaton that is non-deterministic. To see this, note that in the original pattern, the value of the `Sz` variable determines the size of the second segment, which in turn determines the

[1] In ERLANG, variables begin with a capital letter or an underscore, and are possibly followed by a sequence of letters, underscores and digits. A leading underscore in a variable name is typically used to indicate that the variable has only a single use.

Table 3. *Values for* Binary *and matchings for variable* X *in Example 3*

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*

start of the third and last segment. The only way to avoid this form of non-linearity is to "flatten" the binary pattern as in the code shown below:

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

It is a general requirement that binary pattern matching is deterministic. This in turn implies that sizes of segments can be determined by a left-to-right traversal, as in the pattern of Example 2. For example, the following code (where Sz is a variable) is not legal:

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

Naturally, pattern matching against a binary can occur in a function head or in an ERLANG case statement just like any other matching operation. This is illustrated with the following example.

Example 3
Consider the case statement

```
case Binary of
  <<42:8/integer, X/binary>> -> handle1(X);
  <<Sz:8, V:Sz/integer, X/binary>> when Sz > 16 -> handle2(V,X);
  <<_:8, X:16/integer, Y:8/integer>> -> 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 branch of the case statement, 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 have a value greater than 16. Notice that this is both a non-linear and a guarded binary pattern. Finally, if Binary is exactly 32 bits long, X will be bound to an integer consisting of the second and third bytes of the Binary (taken in big-endian order). If neither of the patterns match, the whole match expression will fail. Three examples of matchings and a failure to match using this code are shown in Table 3.

The following two examples show how endianess and signedness specifiers impact binary pattern matching.

Example 4
If X and Y are unbound variables, the matching:

```
<<X:16/integer-big>> = <<0, 42>>
```

results in the binding `X = 42` as the eight low bits of X are 42, while the matching:

```
<<Y:16/integer-little>> = <<0, 42>>
```

results in the binding `Y = 10752` (i.e., 42 * 256) since 42 now appears in the eight high bits.

Example 5
If X and Y are unbound variables, the code:

```
<<X:8/integer-unsigned>> = <<255>>,
<<Y:8/integer-signed>> = <<255>>
```

results in the binding `X = 255, Y = -1`.

Specifiers in the rest of the article. For simplicity of presentation, only `integer` and `binary` specifiers will be used in the rest of this article. Moreover, when a `binary` type specifier is used, we will never specify a size and binary segments will be matched against the complete remaining part of the binary. We will not specify the signedness or the endianess of `integer` segments either; such segments will be considered with their bytes in big-endian order and as unsigned. We also assume that the programs are type-correct.

3 Using binaries for network protocols: some examples

The bit syntax was introduced into Erlang to simplify network protocol implementation. To show that the syntax for manipulating binaries through pattern matching is indeed well-suited for common network protocol programming tasks, we give some examples of how the bit syntax is used in this domain.

Example 6
The function in Fig. 1 accepts IPv6 packets and IPv4 packets without options. It calls the function that is applicable depending on the value of the protocol field. It also acts as a packet filter, making sure that only packets with a certain source and destination IP address are processed and ignoring all the rest.

The binary patterns in the case statement could have been written more succinctly, for example the first pattern could have been written as

```
<<69:8, _:64, ?TCP:8, _Checksum:16, SrcIP:32, DstIP:32, Payload/binary>>
```

but we show each field in the IP packet header explicitly to highlight how easy and natural it is to specify the structure of IP packets and filter them using the binary syntax.

For example, the first four bits of the IP packet header indicate which version of the protocol is used. In this example, the first two patterns can match version 4

```
-define(TCP,6).
-define(UDP,17).
-define(IP_VER4,4).
-define(IP_VER6,6).

filter(Bin, SrcIP, DstIP) ->
  case Bin of
    <<?IP_VER4:4, 5:4, _ToS:8, _ToL:16,
      _Id:16, _Flags:3, _FlagOffset:13,
      _TTL:8, ?TCP:8, _Checksum:16,
      SrcIP:32, DstIP:32, Payload/binary>> -> tcp(Payload);
    <<?IP_VER4:4, 5:4, _ToS:8, _ToL:16,
      _Id:16, _Flags:3, _FlagOffset:13,
      _TTL:8, ?UDP:8, _Checksum:16,
      SrcIP:32, DstIP:32, Payload/binary>> -> udp(Payload);
    <<?IP_VER6:4, _TrafficClass:8, _FlowLabel:20,
      _PayloadLength:16, ?TCP:8, _HopLimit:8,
      SrcIP:128, DstIP:128, Payload/binary>> -> tcp(Payload);
    <<?IP_VER6:4, _TrafficClass:8, _FlowLabel:20,
      _PayloadLength:16, ?UDP:8, _HopLimit:8,
      SrcIP:128, DstIP:128, Payload/binary>> -> udp(Payload);
    _OtherBin -> ok   %% ignore everything else
  end.
```

Fig. 1. Filtering IPv4 and IPv6 packets using the bit syntax.

packets with a header length of 5 words (i.e. a header which contains no optional fields). The first pattern matches the packet if the protocol field contains the value of the TCP macro (defined as 6). This indicates that the payload contains TCP data. The second pattern matches the packet if the protocol field contains the value of the UDP macro (defined as 17) which indicates that UDP is used.

The third and fourth patterns match IPv6 packets. The third pattern matches packets with TCP payloads while the fourth matches packets with UDP payloads.

The filtering done in this example can easily be extended to implement a packet filter which for example considers the port fields in the UDP header if the payload contains UDP data or the TCP options if the payload contains TCP data.

The next example is also related to IP processing. It is a function which can be used to check that the checksum of an IP packet header is correct. It illustrates how the bit syntax can be used to write functions which operate on binaries in the same way that functional programmers typically write functions which operate on lists.

Example 7
The `is_correct_checksum/1` function (Fig. 2) calculates the checksum of an IP packet header and compares it with the value in the checksum field. If the checksums are equal it will return true, otherwise it will return false. (In Erlang, `==` is the built-in term equality operator, `band` is the bitwise and operator, and `bsr` is the operator which shifts bits right a number of positions.)

```
is_correct_checksum(<<InitHeader:80,ChkSum:16,RestHeader/binary>>) ->
  ChkSum == calculate_checksum(<<InitHeader:80,RestHeader/binary>>, 0).

calculate_checksum(<<N:16,Rest/binary>>, Acc) ->
  calculate_checksum(Rest, Acc+N);
calculate_checksum(<<>>, Acc) ->
  two_byte_checksum(Acc).

two_byte_checksum(Acc) when Acc > 16#ffff ->
  two_byte_checksum((Acc band 16#ffff) + (Acc bsr 16));
two_byte_checksum(Acc) ->
  Acc.
```

Fig. 2. IP packet header checksum check.

```
extract_tlb(Bin) ->
  case Bin of
    <<4:8, Length1:8, 0:8, Tag1:16, B1/binary>> -> {Tag1, Length1, B1};
    <<4:8, Length2:8, Tag2:8, B2/binary>>       -> {Tag2, Length2, B2};
    <<5:8, 0:8, Tag3:16, Length3:8, B3/binary>> -> {Tag3, Length3, B3};
    <<5:8, Tag4:8, Length4:8, B4/binary>>       -> {Tag4, Length4, B4}
  end.
```

Fig. 3. Function used as a running example.

The last example in this section is adapted from a program which checks configuration options for the Point-to-Point Protocol (PPP). It has been simplified in order to be used as a running example in the rest of this article and is shown in Fig. 3.

4 Binary pattern matching definitions

Assuming the usual definition of when two non-binary terms (integers, compound terms,...) match, we now turn our attention on how binary pattern matching expressions can be efficiently compiled. A binary pattern matching is defined by a binary term to be matched and a set of binary patterns, which is ordered according to their (usually textual) priority.

In a binary pattern matching compiler, each binary pattern b_i consist of a list of segments $[seg_1, \ldots, seg_n]$ and is associated with a success label (denoted by $SL(b_i)$) which specifies the success continuation.[2]

[2] For simplicity, we keep the fail labels implicit; they are determined by the priority of the binary patterns $\{b_1, \ldots, b_k\}$ as follows:

$$FL(b_i) = b_{i+1}, 1 \leqslant i \leqslant k-1 \text{ and } FL(b_k) = \mathit{failure}.$$

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 a positive 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 (written 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 Example 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 a positive integer constant and is defined as $p_i = \sum_{j=1}^{i-1} s_j$. The presence, however, 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 multiset of size specifiers in preceding segments whose values are not static constants. When V contains just one element (i.e., $V = \{\mathtt{S}\}$), we slightly abuse notation and write V simply as `S`.

Example 8
The binary pattern of Example 2 is represented as

$$[\langle \mathtt{Sz}, \mathtt{integer}, 0, 8 \rangle, \langle \mathtt{Vsn}, \mathtt{integer}, 8, \mathtt{Sz} \rangle, \langle \mathtt{Msg}, \mathtt{binary}, 8 + \mathtt{Sz}, \_ \rangle].$$

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 test and a match test. Each match test includes an associated read action which is to be performed before the actual match test. These notions 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

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

The size test, given a binary b, 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, note that although positions and sizes might not be static constants, in type-correct programs, they are always positive integers at runtime. Thus the second argument of a size test will always be a positive integer constant at runtime.

Definition 2 (*Read action*)
For a segment $\langle v, t, p, s \rangle$, the corresponding read action (denoted by $\mathtt{read}(p, s, t)$) 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.

```
b1 = {size(⩾,40),   match(4,read(0,8,int)), match(Length1,read(8,8,int)),
                    match(0,read(16,8,int)), match(Tag1,read(24,16,int)),
                    match(B1,read(40,_,bin))}
b2 = {size(⩾,24),   match(4,read(0,8,int)), match(Length2,read(8,8,int)),
                    match(Tag2,read(16,8,int)), match(B2,read(24,_,bin))}
b3 = {size(⩾,40),   match(5,read(0,8,int)), match(0,read(8,8,int)),
                    match(Tag3,read(16,16,int)),
                    match(Length3,read(32,8,int)), match(B3,read(40,_,bin))}
b4 = {size(⩾,24),   match(5,read(0,8,int)), match(Tag4,read(8,8,int)),
                    match(Length4,read(16,8,int)), match(B4,read(24,_,bin))}
```

Fig. 4. Optimized action sequences for the binary patterns of Fig. 3.

Definition 3 (*Match test*)
For a segment $\langle v, t, p, s\rangle$, a match test `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 sub-terms of r.

Example 9
The action sequence for the third binary pattern in the case statement of our running example (Fig. 3) is shown below. (For succinctness, we have abbreviated the `integer` and `binary` type specifiers in read actions as `int` and `bin`, respectively.)

```
b3 = {size(⩾, 8), match(5,read(0,8,int)),
      size(⩾, 16), match(0,read(8,8,int)),
      size(⩾, 32), match(Tag3,read(16,16,int)),
      size(⩾, 40), match(Length3,read(32,8,int)),
      match(B3,read(40,_,bin))}
```

Note that this action sequence is sub-optimal. Size tests which are implied by other ones can be removed. When this is done, both for b_3 and for the other binary patterns of our running example, we get the four action sequences shown in Fig. 4.

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

The following definitions will also come in handy.

Definition 4 (*Static size equality*)
Two sizes s_1 and s_2 are statically equal ($s_1 = s_2$) 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 ($p_1 = p_2$) 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 $ra_1 = \texttt{read}(p_1, s_1, t_1)$ and $ra_2 = \texttt{read}(p_2, s_2, t_1)$ are statically equal ($ra_1 = ra_2$) 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 $st = \texttt{size}(op, p + s)$, where op is one of $\{=, \geqslant\}$, is compatible with a binary b (denoted by $st \sqsubseteq b$) 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 $ra = \texttt{read}(p, s, t)$ be a read action. A match test $mt = \texttt{match}(v, ra)$ is compatible with a binary b (denoted by $mt \sqsubseteq b$) if the sub-binary of size s starting at position p of b when read as a term of type t by ra (or more generally by a read action which is statically equal to ra) 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 appearance.

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 (*Pattern priority*)
A pattern $b_j \in B$ has higher priority than a pattern $b_i \in B$ if b_j precedes b_i in B.

Definition 11 (*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 pattern $b_j \in B, j < i$ of higher priority.

5 Adaptive pattern matching on binaries using a tree automaton

5.1 The basic algorithm

The construction of the decision tree automaton (tree automaton for short) begins with a set of k binary patterns ordered by their (usually textual) priority which have been transformed to corresponding action sequences $B = \{b_1, \ldots, b_k\}$. The basic construction algorithm, shown in Fig. 5, builds the tree automaton for B and returns its start node. Each node of the tree automaton consists of an action and two branches (a success and a failure branch) to its children nodes. In interior nodes,

```
Procedure BuildTreeAutomaton($B$)
 1. $u$ := new_tree_node()    // all fields of $u$ are initialized to NULL
 2. if $B = \emptyset$ then
 3.   $u$.action := $failure$
 4. else
 5.   $b_i$ := the action sequence of the highest priority pattern in $B$
 6.   if current_actions($b_i$) $= \emptyset$ then
 7.     $u$.action := goto($SL(b_i)$)    // the success label of $b_i$
 8.   else
 9.     $a$ := select_action($B$)
10.     $u$.action := $a$
11.     $B_s$ := prune_compatible($a, B$)
12.     $u$.success := BuildTreeAutomaton($B_s$)
13.     $B_f$ := prune_incompatible($a, B$)
14.     $u$.fail := BuildTreeAutomaton($B_f$)
15. return $u$
```

Fig. 5. Construction of the tree automaton.

the action is a test. In leaf nodes, the action is a jump (*goto*) to 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. The set B_f consists of action sequences from B that do not contain a. These two sets determine how the tree automaton is constructed. 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 automaton is quite straightforward. Each 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 *failure* leaf is created (lines 2–3 of Fig. 5). When there are still patterns which can match, the action sequence of the highest priority pattern ($b_i \in B, 1 \leqslant i \leqslant k$ such that $b_j \notin B, j < i$) 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 in the previous paragraph. 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).

The select_action procedure controls the traversal order of patterns, making the pattern matching adaptive. It is discussed in section 5.4. The prune_* procedures can be more effective in the amount of pruning that they perform than naïvely constructing the B_s and B_f sets as described above. This issue is discussed in section 5.3.

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 appropriate guard actions to the action sequences and to the actions of decision tree nodes. For example, for the second pattern of Example 3 which is guarded by a `Sz > 16` test, rather than generating a node with a $goto(SL(b_2))$ action when exhausting the actions of b_2, a new node is created whose action is the guard test $\mathtt{guard}(\mathtt{Sz}, >, 16)$, its success branch is the node with action $goto(SL(b_2))$ and its failure branch is obtained by calling $\mathtt{BuildTreeAutomaton}(B \setminus \{b_2\})$.

5.2 Complexity characteristics

Regarding the size of the resulting decision tree, the worst case for this algorithm is when no conclusions can be drawn to prune actions and patterns from B. If k is the number of patterns and n_i is the number of actions in each action sequence, then the size of the constructed tree automaton is:

$$\sum_{i=1}^{k} \prod_{j=1}^{i} (n_i)$$

which is $O(n_{max}^k)$ where n_{max} is the maximum number of actions (segments) in a pattern, i.e. it is exponential in the number of patterns. The time complexity for the worst case path through this tree is linear in the total number of segments.

5.3 Basic 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:

$\mathtt{prune\_compatible}(a, B)$ Removes all actions from action sequences in B which can be proved to be compatible with any binary b such that $a \sqsubseteq b$ and all action sequences that contain an action which can be proved to be incompatible with any binary b such that $a \sqsubseteq b$.

$\mathtt{prune\_incompatible}(a, B)$ Removes all actions from action sequences in B which can be proved to be compatible with any binary b such that $a \not\sqsubseteq b$ and all action sequences that contain an action which can be proved to be incompatible with any binary b such that $a \not\sqsubseteq b$.

5.3.1 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

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

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

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

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

Fig. 6. Size pruning rules.

structured terms. If equality ($=$) 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 $\geqslant$ 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 set up rules that allow us to infer the compatibility or incompatibility of a 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 $c + V$ 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 error will occur).

Definition 12 (*Statically comparable size expressions*)
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 by $se_1 = se_2$) if $c_1 = c_2$ and V_1 is the same multiset as V_2;
- se_1 is statically larger than se_2 ($se_1 > se_2$) if $c_1 > c_2$ and V_1 is a superset of V_2;
- se_1 is statically larger or equal to se_2 ($se_1 \geqslant se_2$) 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 from se_2 ($se_1 \neq se_2$) 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 $st_i \in B$ 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 Fig. 6(a) presents the conclusions 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 $st_i \not\sqsubseteq b$. The rules in Fig. 6(b) describe 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.

Example 10

To illustrate size test pruning, let $st = \texttt{size}(=, 24{+}\texttt{Sz})$ and $B = \{b_1, b_2, b_3, b_4\}$ where:

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

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 $b'_1 = \{a_{1,2}, \dots, a_{1,n_1}\}$, and $b'_2 = \{a_{2,2}, \dots, a_{2,n_2}\}$. Why the size test st is removed from b_1 should be obvious. In b_2, the size test $\texttt{size}(\geqslant, 24)$ is implied by st (see the third row of Table 6(a)) and is removed. Sequences b_3 and b_4 each contain a size test which fails if st succeeds (this is found by looking at rows six and four of Table 6(a)) and are pruned. We also have that $\texttt{prune\_incompatible}(st, B) = \{b_2, b_3, b_4\}$.

5.3.2 Match test pruning

A simple form of match test pruning can be based on the concept of similarity of match tests. 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:

$$\begin{aligned} mt_1 \sqsubseteq b &\Rightarrow mt_2 \sqsubseteq b \\ mt_1 \not\sqsubseteq b &\Rightarrow mt_2 \not\sqsubseteq b \end{aligned}$$

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 Section 5.6.2 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.

5.4 Adaptive selection of actions

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 cannot 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 minimizing the size of a binary decision tree is an NP-complete problem (Hyafil & Rivest, 1976), we employ heuristics. One such heuristic is to select actions which make the size of the success subtree of a node small. Such actions, called eliminators, are defined below.

Definition 13 (*Eliminators*)
Let $B = \{b_1, \ldots, b_k\}$ be an ordered 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 maximal eliminator if it is an eliminator of m sequences and for all $l > m$ there do not exist eliminators of l sequences.

So we are looking for maximal eliminators, ideally 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 decision 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 following heuristics can be used. Some of them reduce the size of the tree, and some reduce the time needed to find a matching.

Eliminator A maximal eliminator is chosen. As a tie-breaker, a top-down, left-to-right order of selecting maximal 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 Scott & Ramsey (2000).

We evaluate the effects of these heuristics on a set of benchmarks in Section 7.2.

5.5 Example of building a tree automaton

Having described all procedures used in the `BuildTreeAutomaton` algorithm (Fig. 5) we show an example of how the algorithm actually works. Consider the piece of code in Fig. 3 and the corresponding action sequences in Example 9. Note that none of the variables that are being matched are bound or used in a non-linear way, which means that all match tests involving variables will succeed. This allows us to postpone such match tests until we have determined which pattern matches the incoming binary. (In other words, until we have found a match we need not consider these match tests. Also, when the match is found the nodes containing these match tests need success branches only.) So, the action sequences to consider at the beginning of the tree construction are the following ones:

$$\begin{aligned}
&b_1=\{\texttt{size}(\geqslant,40),\ \texttt{match}(4,\texttt{read}(0,8,\texttt{int})),\ \texttt{match}(0,\texttt{read}(16,8,\texttt{int}))\}\\
&b_2=\{\texttt{size}(\geqslant,24),\ \texttt{match}(4,\texttt{read}(0,8,\texttt{int}))\}\\
&b_3=\{\texttt{size}(\geqslant,40),\ \texttt{match}(5,\texttt{read}(0,8,\texttt{int})),\ \texttt{match}(0,\texttt{read}(8,8,\texttt{int}))\}\\
&b_4=\{\texttt{size}(\geqslant,24),\ \texttt{match}(5,\texttt{read}(0,8,\texttt{int}))\}
\end{aligned}$$

If we use the left-to-right heuristic we should first select the action $\texttt{size}(\geqslant,40)$. Using the rules in Table 6(a), we find out that $\texttt{size}(\geqslant,40)$ and $\texttt{size}(\geqslant,24)$ are compatible with the binary in the `prune_compatible` procedure. This means that B_s will contain the following action sequences:

$$\begin{aligned}
&b_1=\{\texttt{match}(4,\texttt{read}(0,8,\texttt{int})),\ \texttt{match}(0,\texttt{read}(16,8,\texttt{int}))\}\\
&b_2=\{\texttt{match}(4,\texttt{read}(0,8,\texttt{int}))\}\\
&b_3=\{\texttt{match}(5,\texttt{read}(0,8,\texttt{int})),\ \texttt{match}(0,\texttt{read}(8,8,\texttt{int}))\}\\
&b_4=\{\texttt{match}(5,\texttt{read}(0,8,\texttt{int}))\}
\end{aligned}$$

For the `prune_incompatible` procedure we find that $\texttt{size}(\geqslant,40)$ is incompatible with the binary and that we cannot say anything about $\texttt{size}(\geqslant,24)$. The B_f set will thus contain the following action sequences:

$$\begin{aligned}
&b_2=\{\texttt{size}(\geqslant,24),\ \texttt{match}(4,\texttt{read}(0,8,\texttt{int}))\}\\
&b_4=\{\texttt{size}(\geqslant,24),\ \texttt{match}(4,\texttt{read}(0,8,\texttt{int}))\}
\end{aligned}$$

To show how match pruning works, we now show the result from applying the `BuildTreeAutomaton` procedure to B_s. Suppose that the `select_action` procedure chooses $\texttt{match}(4,\texttt{read}(0,8,\texttt{int}))$ as selected action. If this test is compatible with a binary we know from the rules in Section 5.3.2 that $\texttt{match}(5,\texttt{read}(0,8,\texttt{int}))$ is incompatible with that binary. This means that the `prune_compatible` procedure would return the following action sequences:

$$\begin{aligned}
&b_1=\{\texttt{match}(0,\texttt{read}(16,8,\texttt{int}))\}\\
&b_2=\{\}
\end{aligned}$$

For the `prune_incompatible` procedure on the other hand we would end up with the following action sequences:

$$\begin{aligned}
&b_3=\{\texttt{match}(0,\texttt{read}(8,8,\texttt{int}))\}\\
&b_4=\{\}
\end{aligned}$$

When the `BuildTreeAutomaton` procedure is complete, we end up with the decision tree automaton shown in Fig. 7.

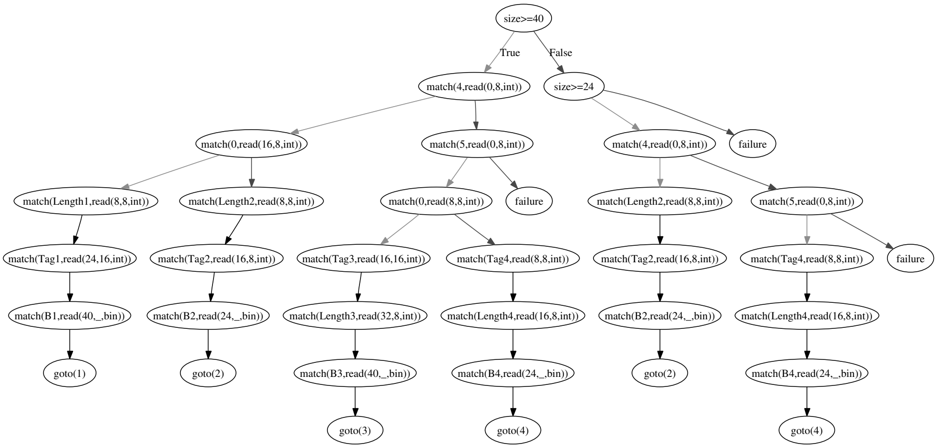


Fig. 7. The tree automaton created for the program of Fig. 3.

5.6 Optimizations

The basic decision tree construction algorithm presented so far makes no attempt to reduce the size of the resulting tree automaton. We therefore present three kinds of optimizations that can decrease its size; in practice they often do so quite effectively.

5.6.1 Turning the tree automaton into a directed acyclic graph

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 finite state automaton minimization techniques to create the optimal DAG. This might however be impractical, since it requires that a tree automaton of possibly exponential size is first constructed. Instead, we use a concept similar to memoization to construct the DAG directly. We simply remember the results we got from calling the `BuildTreeAutomaton` procedure, and if the procedure is called again with the same input argument, we simply return the subtree that was constructed at that time.

We show the directed acyclic graph that we create for our running example in Fig. 8. This optimization alone decreases the number of nodes from 35 down to 25.

Note that 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 pruning the selected actions perform, the harder it is to share subtrees. In our experience however, turning the tree into a DAG is an effective size-reducing optimization in practice.

5.6.2 Pruning based on interference of match tests

Recall that basic pruning based on match tests, introduced in Section 5.3.2, takes place when two match tests contain read actions which are statically equal. We

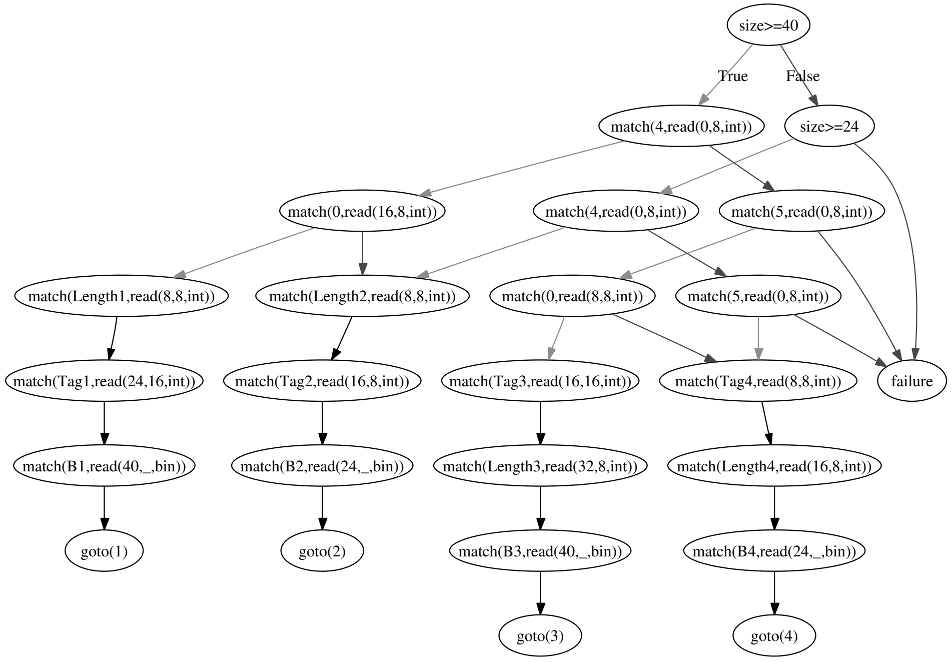


Fig. 8. The directed acyclic graph corresponding to the tree of Fig. 7.

can increase the amount of pruning performed based on match tests by taking interferences between match tests into account. The idea is easy to illustrate with an example.

Example 11
In the binary patterns $b_1 =$ `<<Sz:4, 0:12, X:Sz>>` and $b_2 =$ `<<255:8, ...>>` there do not exist any statically equal read actions in match tests. It is, however, 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 14 (*Interference*)
Let p_1 and p_2 be statically known positions where $p_1 \leqslant p_2$. Also, let s_1 and s_2 be statically known sizes. We say that two 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 statically known. Let us denote the common bits of v_1 and v_2 by v_1' and v_2', respectively.

Definition 15 (*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. Without loss of generality, let $p_1 \leqslant p_2$. We say that mt_1 encloses mt_2 (denoted $mt_1 \supseteq mt_2$) if $p_1 + s_1 \geqslant p_2 + s_2$.

Now consider two match tests mt_1 and mt_2 which interfere and let 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. The last rule can be used to guide the pruning in the `prune_incompatible`(mt_1, B) procedure.

This optimization is particularly important for network protocol applications such as packet classification. In such applications it is typical that some patterns match on the first 8 bits of the IP address, others match on the first 24 bits, and others on the entire address.

5.6.3 Factoring read actions

To ease exposition of the main ideas, we have thus far presented read actions as tightly coupled with match tests 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 renders 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 for read actions ra_k that are statically equal to a read action ra which has already been performed. Then the result of ra can be saved in some temporary register, and each of the ra_k actions can then be replaced by a use of that register. (This is a standard compiler optimization called *global common subexpression elimination.*) Our experience is that in practice this caching read values into registers significantly reduces the time to perform binary pattern matching.

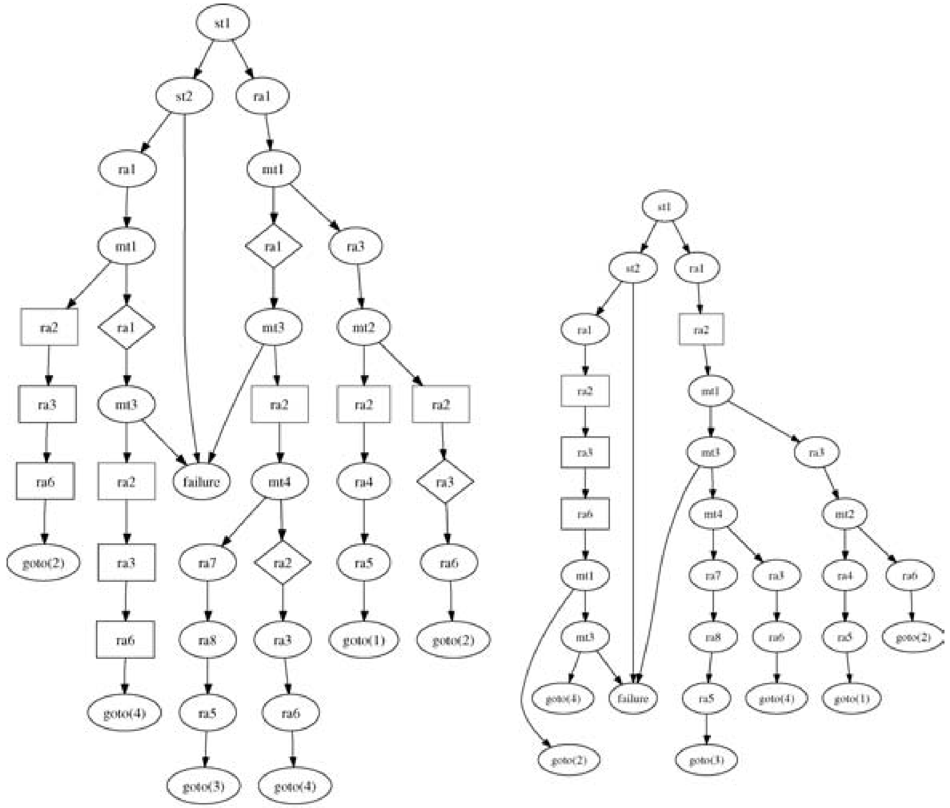
Also, to reduce code size, other standard compiler techniques like *code hoisting* can be used to move a read action to a node in the tree automaton where a statically equal read action will be performed on all paths from that node to a leaf containing a $goto(SL(b))$ action. These read actions can then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

To illustrate the results of this optimization we will show its effect on our running example. In order to do this we need to separate the read actions from the match tests. To do this in a more succinct way, we will use shorter names for the actions; these names are shown in Table 4.

To show the total effect of the optimization we show the tree automaton where the read actions have been separated from the match tests and where all read actions are present in the automaton, even those that are only used in match tests where

Table 4. *Short names for actions in our running example*

$st_1 = \texttt{size}(\geqslant, 40)$	$st_2 = \texttt{size}(\geqslant, 24)$
$ra_1 = \texttt{read}(0, 8, \texttt{int})$	$ra_2 = \texttt{read}(8, 8, \texttt{int})$
$ra_3 = \texttt{read}(16, 8, \texttt{int})$	$ra_4 = \texttt{read}(24, 16, \texttt{int})$
$ra_5 = \texttt{read}(40, \_, \texttt{bin})$	$ra_6 = \texttt{read}(24, \_, \texttt{bin})$
$ra_7 = \texttt{read}(16, 16, \texttt{int})$	$ra_8 = \texttt{read}(32, 8, \texttt{int})$
$mt_1 = \texttt{match}(4, ra_1)$	$mt_2 = \texttt{match}(0, ra_3)$
$mt_3 = \texttt{match}(5, ra_1)$	$mt_4 = \texttt{match}(0, ra_2)$



(a) The automaton before read factoring; read actions in diamond shaped boxes will be removed because they are dominated by a statically equal read action; nodes in square boxes will be hoisted.

(b) The automaton after read factoring; nodes in square boxes have been hoisted. Also note the square node containing ra2 immediately below r1. This node serves the rôle of the three nodes containing ra2 on the right half of Fig.9(a).

Fig. 9. Tree automata before and after read factoring.

the term to be matched is an unbound variable. This automaton, containing all read actions, is shown in Fig. 9(a); the result after the read factoring optimizations are performed is shown in Fig. 9(b).

6 A space conservative approach

The main drawback of the decision tree automaton approach is that the size of the automaton can be exponential in the number of patterns. In most applications the optimizations of section 5.6 are quite effective and make the size of the tree automaton manageable, but decision tree automata provide no polynomial space guarantees and in pathological cases the size of the automaton can explode. When code space is at a premium, as in embedded controllers, a space conservative approach which can avoid code explosion might be called for.

The obvious choice in this case would be to use a *backtracking automaton*, similar to those proposed by Augustsson (1985) for structured terms, to perform the pattern matching. In backtracking automata, each of the actions of every pattern appears only once, but on the other hand, there is no sharing of similar actions across different patterns. Since it is often typical in our context to have statically equal actions which belong to several action sequences, in practice the size of the backtracking automaton without sharing of these statically equal actions can be similar to that of the tree automaton. Moreover, backtracking automata have the disadvantage that they do not provide polynomial execution time guarantees.

We want to do better than that. For that reason, in this section we introduce the concept of a *guarded sequential automaton.*

6.1 Guarded sequential automata: Properties and mode of operation

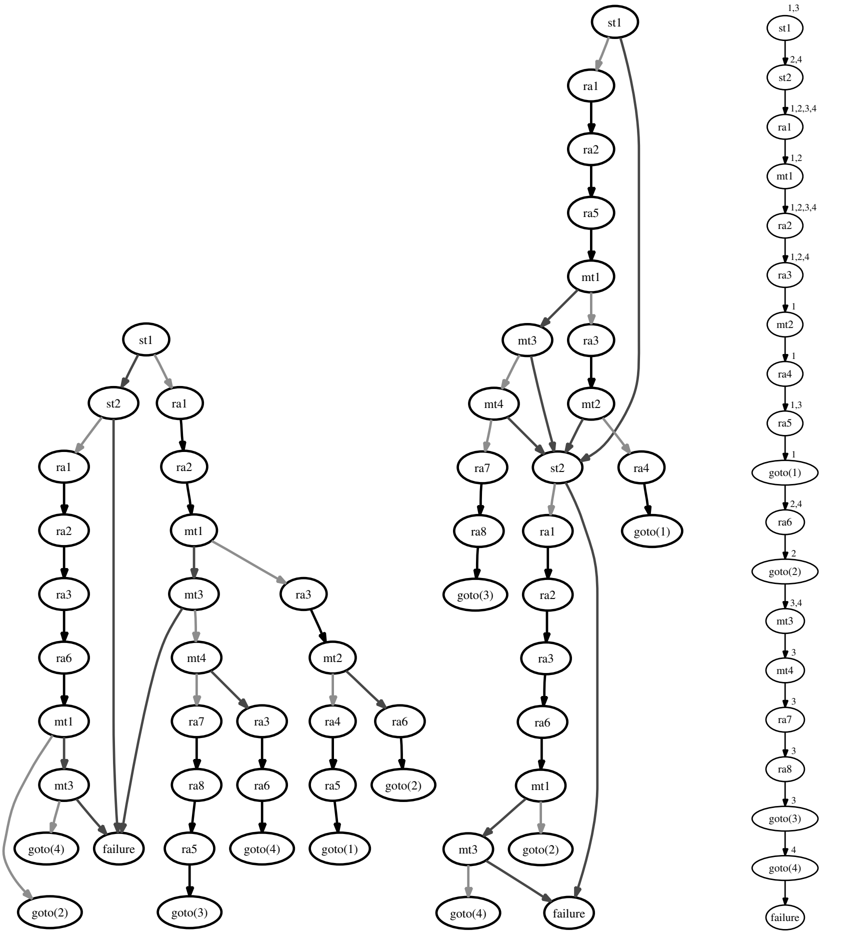
For our space conservative approach we would like to produce an automaton with the following properties:

1. Its space requirements are minimal in the sense that each statically distinct action appears at most once in the automaton.
2. During execution, no statically distinct action is performed more than once.

Before describing a binary pattern matching approach which achieves these two properties, let us re-examine the various kinds of automata and compare their modes of operation. Recall the program that we have used as our running example:

```
extract_tlb(Bin) ->
  case Bin of
    <<4:8, Length1:8, 1:8, Tag1:16, B1/binary>> -> {Tag1, Length1, B1};
    <<4:8, Length2:8, Tag2:8, B2/binary>>       -> {Tag2, Length2, B2};
    <<5:8, 1:8, Tag3:16, Length3:8, B3/binary>> -> {Tag3, Length3, B3};
    <<5:8, Tag4:8, Length4:8, B4/binary>>       -> {Tag4, Length4, B4}
  end.
```

The tree automaton produced by the `BuildTreeAutomaton` algorithm for these binary patterns using the left-to-right heuristic and read hoisting is shown in Fig. 10(a). The deterministic form of a backtracking automaton we could create using standard techniques is shown in Fig. 10(b). The unoptimized form of the automaton produced by the space conservative approach, which we will call a *guarded sequential automaton*, is shown in Fig. 10(c).



(a) Tree (a) Backtracking (a) Guarded sequential

Fig. 10. Three pattern matching automata for our running example.

It is easy to describe how the decision tree and backtracking automata are used to find the pattern which matches the binary. When there are two edges out of a node, one of them is taken when the action in the node is compatible with the binary and the other one when the action is incompatible. A match has been found when we reach a node containing a *goto* action. When the *failure* node is reached, we know that no pattern matches the binary.

How the guarded sequential automaton operates is not so clear. Because the actions of various action sequences are interspersed, in order to use it for binary pattern matching we must answer the following questions:

1. How do we determine that a match has been found?
2. How do we bypass execution of unsafe read actions and inappropriate *goto* actions?

To determine that we have found a match with pattern b_i, we need to know that all relevant actions of b_i have been performed and all of them are compatible with the binary we are matching. Also, in accordance with Definition 11, we need to ensure that none of the sequences with higher priority is an instance of the binary. To remember which binary patterns from $B = \{b_1, \ldots, b_k\}$ can still match with a binary b, we can associate a boolean variable with each b_i. Let us call this variable the π-variable of b_i (or π_i for short). These variables will be initialized to *true* and each π_i will hold this value as long as all of the actions of b_i that have been performed so far are compatible with b. As soon as the guarded sequential automaton performs an action that belongs to b_i which is incompatible with b, the value of π_i will be changed to *false*. If π_i is still *true* when we have performed all of the actions that belong to b_i, we know that b_i is an instance of b. If at any point π_i becomes *false*, we know that b_i cannot be an instance of b and thus cannot match b.

Before describing how to bypass unsafe read actions, we first need to define when a read action is safe.

Definition 16 (*Safety of read actions*)
In the context of a binary pattern matching between a set of action sequences $B = \{b_1, \ldots, b_k\}$ and a binary b, a read action *ra* is safe if there exists an action sequence $b_i \in B$ such that $ra \in b_i$ and for all size tests $st \in b_i$ such that *st* precedes *ra*, *st* is compatible with b.

Naturally, we not only want to perform read actions when these are safe, but we also want to avoid performing unnecessary actions.

Definition 17 (*Avoidability of actions*)
In the context of a binary pattern matching between a set of action sequences $B = \{b_1, \ldots, b_k\}$ and a binary b, an action a is avoidable if there exists no $b_i \in B$ such that $a \in b_i$ and b_i can match b.

That is, before performing an action we must be able to determine whether it is avoidable at this point in time, in which case we could simply ignore it, but we also have to be certain that it is safe, because only in this case we are allowed to perform that action. So both read and match actions are constrained to appear only on certain positions in the guarded sequential automaton. Thus, to create the automaton, we need a notion of selectability of actions similar to the one discussed in Section 5.4 for constructing the tree automaton.

Definition 18 (*Selectability of actions*)
The condition determining whether an action is selectable depends on its type:

- A size test is selectable when its size expression can be evaluated to a constant.
- A read action is selectable when all of the size tests which precede it in any of the action sequences that the read action belongs have been selected.
- A match test is selectable when the corresponding read actions have been selected.

Naturally, the construction algorithm for the guarded sequential automaton needs to respect the constraint on selectability of actions. Also, it can easily be seen that selectability of read actions implies their safety. We can now describe how the automaton is constructed.

```
Procedure BuildGuardedSequence($B$)
 1. $u$ := new_guarded_node()
 2. if $B = \emptyset$ then
 3.   $u$.action := $failure$
 4. else
 5.   $b_i$ := the highest priority action sequence in $B$
 6.   if current_actions($b_i$) $= \emptyset$ then
 7.     $u$.action := $goto(SL(b_i))^{\pi_i}$
 8.     $B'$ := $B \setminus \{b_i\}$
 9.     $u$.next := BuildGuardedSequence($B'$)
10.   else
11.     $a^{\Pi}$ := select_and_annotate_action($B$)
12.     $u$.action := $a^{\Pi}$
13.     $B'$ := remove_equal($a, B$)
14.     $u$.next := BuildGuardedSequence($B'$)
15. return $u$
```

B, B'	: sets of action sequences
b_i	: *i-th* action sequence
$SL(b_i)$	: the success label of b_i
a^{Π}	: guarded action
π_i	: guard with index i
u	: automaton node

Fig. 11. Construction of the guarded sequential automaton.

6.2 The basic algorithm

Figure 11 shows an algorithm which creates this guarded sequential automaton. Each state in the automaton is represented by an action annotated by a set of π-variables. There is one variable for each sequence that the action is a member of.

Whenever the highest priority action sequence is exhausted, a guarded *goto* action node is created (lines 5–9). Otherwise, the select_and_annotate_action(B) procedure chooses one of the actions in B, provided it is selectable according to Definition 18, and annotates it with all the π-variables of action sequences that contain a statically equal action. This creates a new automaton node.

The remove_equal(a, B) procedure removes the action a from all action sequences in B. Note that if the action a has been annotated with n π-variables in line 11 of the algorithm (i.e., $|\Pi| = n$), exactly n actions will be removed from B in line 13.

The result of the BuildGuardedSequence procedure is an automaton where each node contains an action annotated with the π-variables corresponding to action sequences the action belongs to. The π-variables guard the execution of the action; an action will be performed only if at least one of these variables has the value *true*. If a node's action fails, all the π-variables which annotate this node will be set to *false*. Unless a final state has been reached, both upon success and failure of an action, the guarded sequential automaton moves to the next state.

Nodes containing a *goto* action are considered accepting states of the automaton; i.e., states where a match has been found. Note that each $goto(SL(b_i))$ action is also annotated – and therefore guarded – by the π-variable associated with the action sequence b_i. Thus, it is not possible to perform an inappropriate *goto* action when using the guarded sequential automaton. If the *failure* node is reached, the matching has failed.

6.3 Example of building a guarded sequential automaton

To show how the BuildGuardedSequence algorithm constructs the guarded sequential automaton shown in Fig. 10(c), consider the action sequences of the four binary patterns of our running example with action names abbreviated as in Table 4.

$$\begin{aligned}
b_1 &= \{st_1, ra_1, mt_1, ra_2, ra_3, mt_2, ra_4, ra_5\} \\
b_2 &= \{st_2, ra_1, mt_1, ra_2, ra_3, ra_6\} \\
b_3 &= \{st_1, ra_1, mt_3, ra_2, mt_4, ra_7, ra_8, ra_5\} \\
b_4 &= \{st_2, ra_1, mt_3, ra_2, ra_3, ra_6\}
\end{aligned}$$

Note that for these $k = 4$ sequences, the total number of actions is $n = 28$ but the number of distinct actions is $m = 14$.

Let us assume a left-to-right, top-down selection strategy for actions. This causes st_1 to be selected first. The node which is created is annotated with the boolean guard $\pi_1 \vee \pi_3$ since st_1 is present in sequences b_1 and b_3. The next action to consider is ra_1, but this is not a selectable action, since it is present in all sequences and some of them still have unselected size tests which precede this action. As a result, the next action to select is st_2 and the next node is annotated with $\pi_2 \vee \pi_4$ since st_2 is a member of sequences b_2 and b_4. We can now select ra_1. Since ra_1 is present in all four action sequences its node is annotated with $\pi_1 \vee \pi_2 \vee \pi_3 \vee \pi_4$. We continue selecting actions in this manner until b_1 no longer contains any action at which point we insert an appropriate *goto* action annotated with π_1. In the end, we end up with an automaton with the following sequence of guarded actions.

$$\begin{aligned}
\langle\ & st_1^{\pi_1\vee\pi_3},\ st_2^{\pi_2\vee\pi_4},\ ra_1^{\pi_1\vee\pi_2\vee\pi_3\vee\pi_4},\ mt_1^{\pi_1\vee\pi_2},\ ra_2^{\pi_1\vee\pi_2\vee\pi_3\vee\pi_4}, \\
& ra_3^{\pi_1\vee\pi_2\vee\pi_4},\ mt_2^{\pi_1},\ ra_4^{\pi_1},\ ra_5^{\pi_1\vee\pi_3},\ goto(1)^{\pi_1},\ ra_6^{\pi_2\vee\pi_4},\ goto(2)^{\pi_2}, \\
& mt_3^{\pi_3\vee\pi_4},\ mt_4^{\pi_3},\ ra_7^{\pi_3},\ ra_8^{\pi_3},\ goto(3)^{\pi_3},\ goto(4)^{\pi_4}, failure\rangle
\end{aligned}$$

This is the automaton shown in Fig 10(c) which has $m = 14$ ordinary states, $k = 4$ accepting states and one *failure* state.[3] Two of the ordinary states contain size tests, four contain match tests and eight contain read actions. In contrast, if we use the pruning heuristic to create a DAG automaton we would need 23 ordinary states when read hoisting is used, and 19 ordinary states when the backtracking automaton approach is used.

6.4 Complexity characteristics

There are three different costs involved in performing binary pattern matching this way: testing and updating π-variables and performing actions (i.e., read actions, size and match tests).

Proposition 1

If $B = \{b_1, \ldots, b_k\}$ is a set of k action sequences containing a total of n actions then the guarded sequential automaton constructed for these sequences will perform at most $(n + k)$ π-variable guard checks.

[3] Note that in Fig. 10(c) the nodes are annotated by integers rather than boolean expressions, but the correspondence between the two notations should be clear.

Proof
Note that the transitions between consecutive nodes of the guarded sequential automaton form a chain and during operation each node is visited only once.

The automaton contains a total of n π-variable guards annotating its non-accepting nodes. Each of these n variables will be tested at most once to decide whether to perform the action in each non-accepting node.

Each one of its k accepting nodes (i.e. nodes annotated with *goto*s) are guarded with only one π-variable each, and these nodes too will only be visited once. Therefore, the total number of π-variable guard checks is at most $(n+k)$. □

Proposition 2
If B is a set of k action sequences containing a total of n actions then the guarded sequential automaton for B will perform at most $min(\frac{k(k+1)}{2}, n)$ π-variable updates.

Proof
π-variables are updated only if an action fails. If all actions fail, at most n π-variables will be updated as this is the total number of π-variables annotating the action-containing nodes of the automaton.

Also note that at most k actions can fail as at least one π-variable will be changed to *false* when an action fails. Therefore when k actions have failed all variables will have the truth value *false* and no more actions will be performed. The only case when k actions fail is when only one variable is changed from *true* to *false* for each failure.

A variable is set to *false* at a failure whether it is *false* or *true*. In the worst case scenario, the number of variables which are set to *false* at each failure is the number of variables which contain *false* plus one, since only one variable is changed from *true* to *false* at each failure. Since the number of variables which contain *false* is equal to the number of failures which have occurred, the bound on updated variables for failure i is $1+(i-1)$. There will be at most k failures and thus we get:

$$\sum_{i=1}^{k} 1 + (i-1) = \sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

Since the number of updates is limited by both these numbers the minimum of these numbers is a limit for the number of π-variable updates. □

Proposition 3
For a set of k action sequences $B = \{b_1, \ldots, b_k\}$ which contains a total of m statically distinct actions, at most $m+k$ transitions will be needed before the guarded sequential automaton for B successfully reaches an accepting state.

Moreover, during runtime, at most m actions (i.e., read actions, size and match tests) actions will be performed.

Proof
The total number of nodes in a guarded sequential automaton for a set of m statically distinct actions is m nodes annotated with (read, size and match test) actions, k nodes with *goto*s and one *failure* node which always appears last. □

Let us summarize the results in these three propositions. If T_T, T_U, and T_A are the times it takes to perform a test, an update of a π-variable, and an action respectively, and T is the total time it takes to perform a matching on k action sequences containing a total of n actions out of which m are statically distinct, then we have the following relation:

$$T = (n + k) \times T_T + min\left(\frac{k(k+1)}{2}, n\right) \times T_U + m \times T_A$$

This result indicates that the guarded sequential automaton approach is expedient only if the cost of tests and updates of the π-variables is significantly smaller than the cost of performing the corresponding actions. It is reasonable to expect that this is the indeed the case since the boolean-valued π-variables can each be represented by one bit and tests can be performed in groups of such variables.

6.5 Optimizations

We can actually create a slightly more efficient variant of the guarded sequential automaton by performing the following two kinds of optimizations.

6.5.1 Avoiding unnecessary tests

Sometimes testing the values of the π-variables is unnecessary. One trivial such example is the first time a π-variable is used, since we know that all π-variables are initialized to *true* and their value does not change until we have performed an action in a node guarded by the corresponding variable.

There are two more cases when we can use similar reasoning to avoid tests:

- Suppose that we have two consecutive actions $a_1^{\Pi_1}$ and $a_2^{\Pi_2}$, where Π_1 is a subset of Π_2 (denoted $\Pi_1 \subseteq \Pi_2$). If we find that a_1 is compatible with the binary, then we know that at least one of the variables in Π_1 holds the value *true*. Since $\Pi_1 \subseteq \Pi_2$ and the variables in Π_1 do not change values, then we also know that at least one of the variables in Π_2 holds the value *true*. Thus, we do not have to test the variables in Π_2 at runtime.
- Suppose we have an action $a_1^{\Pi_1}$ and we find out that all of the variables in Π_1 contain the value *false*. We can find this out, either from the test of Π_1, or if a_1 is not compatible with the binary in which case all of the variables in Π_1 will be set to *false*. This allows us to conclude that a transition to an action $a_2^{\Pi_2}$ when $\Pi_2 \subseteq \Pi_1$ will not be possible. In this case we should instead try to transition directly to the next action $a_j^{\Pi_j}$ for which $\Pi_j \nsubseteq \Pi_1$.

This approach to avoiding tests is based on local reasoning about the possible values of the π-variables and is the one we have implemented. It can of course be extended to a full-fledged path sensitive analysis of the possible values of the π-variables, but this could be quite costly since the cost of performing the analysis would be proportional to the total number of paths in the automaton which in turn is exponential in the number of nodes. We want to avoid exponential costs in the

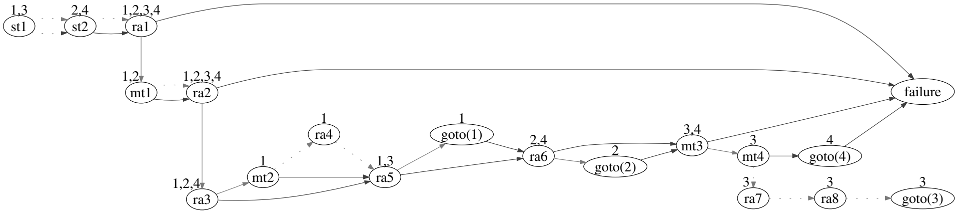


Fig. 12. Guarded automaton where some guard tests are skipped.

guarded sequential automaton approach; which in turn disqualifies the use of a path sensitive analysis.

Note that after performing such optimizations to avoid unnecessary π-variable tests, we no longer have a sequential automaton. Instead we take one of two different transitions from a node. One transition is chosen when the guard succeeds and the action is compatible to the binary we are matching against. The other transition is chosen when either the guard fails or the action is incompatible with the binary we are matching against. The first transition is always to the next node in the sequential automaton, the other can be to any node later in the sequence. Also note that this optimization preserves the characteristics we wanted for our space conservative approach since no new nodes are created and all transitions in the new automaton are from a node earlier in the sequence to a node that is later in the sequence.

The result of applying this optimization to the guarded sequential automaton of Fig. 10(c) is shown in Fig. 12. (Dotted lines denote that the π-variables of the destination node do not need to be tested during these transitions.) It is notable that this optimization removes the need to test the π-variables about half the time that an action is compatible with the binary and that it is possible to skip at least one node more than half the time that a test fails.

6.5.2 Joining match tests

Another possible optimization is to combine match tests which have the same read action. In our running example, the value returned by the read action $\texttt{read}(0, 8, \texttt{int})$ is matched with four in one node and with five in another. If the first match test succeeds, we know that the action sequences that contain a match with five will fail. This means that we can perform the matching more efficiently. The rules for how the π-variables are updated under these circumstances are different from the ordinary case. In this case each possible value is associated with the variables of one or more action sequences. All of the π-variables associated with cases which do not match are set to *false*.

Let $B' = \{b_1 \ldots b_l\}$ be a set of action sequences which match the same read action to a set of l different values and let $\Pi = \pi_1 \cup \ldots \cup \pi_l$ denote the set of π-variables which are associated with B'. We perform the "parallel" matching action if any variable in Π is *true*. If the matching action succeeds with the value of the match test in b_i then all variables in $\Pi \setminus \{\pi_i\}$ are set to *false*. Furthermore, if no value is matched all variables in Π are set to *false*.

This optimization can be very effective particularly if there is one field in the binary which is used as an index to decide which sequence to match. (This is very similar to pattern matching against structured terms in the cases where constructors provide such an index.) The impact of this optimization increases if it is used together with the static analysis since more information will come from one big matching action than from several small matches in a sequence.

6.6 A hybrid approach

Note that the two approaches to binary pattern matching that we have described in section 5 and in this section create their matching automata from the same building blocks. Therefore it is possible to combine them by calling `BuildGuardedSequence` rather than `BuildTreeAutomaton` under some circumstances within the body of the `BuildTreeAutomaton` procedure. This way, one can limit the size of the resulting decision tree automaton while still profiting from the runtime advantages of performing binary pattern matching using decision trees.

6.7 Discussion

The guarded sequential automaton approach to binary pattern matching that we have introduced and described has nice theoretical properties and is quite intriguing. Its advantages are that the size of the automaton is linear in the total number of (non-similar) actions. Also, note that a relatively little machinery is needed to implement it: one variable per binary pattern and a mechanism for testing the truth value of guards. Since the π-variables are boolean-valued they can be represented using a single bit, and guards of nodes can be implemented using bit vectors. This implementation has the additional advantage that testing whether a disjunction of π-variables is *true* boils down to testing whether the bit vector is zero or not. So, the approach is fast and its space requirements are small. More importantly, there is no risk of code explosion which makes the approach of interest for embedded telecom controllers, for instance, where code size is a concern.

On the other hand, a disadvantage is that the runtime cost of finding a match depends linearly on the total number of distinct actions. The fact that some read actions interfere is not exploited in this approach. This is unfortunate since in some applications that optimization alone is very effective.

7 Experimental evaluation

In previous work (Gustafsson & Sagonas, 2002), we have presented a scheme for efficient compilation of BEAM instructions that manipulate binaries to native code.[4] On a set of benchmarks, when executing native code, the speedups range from 20%

[4] BEAM is the virtual machine of the Erlang/OTP (Open Telecom Platform) system. Native code compilation of binaries is available in the Erlang/OTP distribution since October 2002 and the adaptive pattern matching scheme we describe in this article since October 2004; see www.erlang.org.

to four times faster compared with BEAM. The native code compilation scheme of Gustafsson & Sagonas (2002) is the basis of our implementation on top of which we implemented the various binary pattern matching automata approaches described in this article. In this section, we evaluate their code space and runtime performance using standard benchmark programs from the area of packet classification and from actual telecom applications written in ERLANG.

7.1 Packet classification

One of the possible application areas for binary pattern matching is packet classification. That is classifying network packets in order to treat them differently depending on the contents of the packet headers.

Typically packet classification is based on a five tuple of values (Destination IP-address, Source IP-address, Protocol Number, Destination Port, and Source Port). In typical packet classification algorithms (Baboescu & Varghese, 2001; Gupta & McKeown, 2001) these values are first extracted from the packets and the packets are then classified. Our approach does not require any such extraction, nor does it require that the problem can be described as matching on a few distinct header fields. This is possible since the bit syntax is used to match directly on the packets. Therefore our approach is easily extensible to more complex classification rules which e.g., use fields from higher level protocol headers if they are available.

To evaluate the effectiveness of the different approaches for this application we used the ClassBench system to create a set of rules and a set of packets to exercise the rules. ClassBench (Taylor, 2004) is a benchmarking framework for the packet classification area. It allows the user to create synthetic rule sets whose characteristics are determined by a specification file. ClassBench is distributed with several specification files which have been distilled from real rule sets for packet classification. For our first benchmark, we used the `acl1` rule set specification.

We compare three different compilation methods: one which uses a tree automaton, one which uses a guarded sequential automaton and one which uses a backtracking automaton (this is what the BEAM bytecode compiler implements). For each method, we measured the compilation time, the size of the resulting code, and the time it took to classify three million packets for several different numbers of rules. The experiments were run on a 2.0 GHz AMD Athlon64 machine with 1 GByte of memory running Linux. Table 5 shows the raw data obtained. To see the big picture more easily, we also present code sizes and run times in the form of graphs; see Fig. 13.

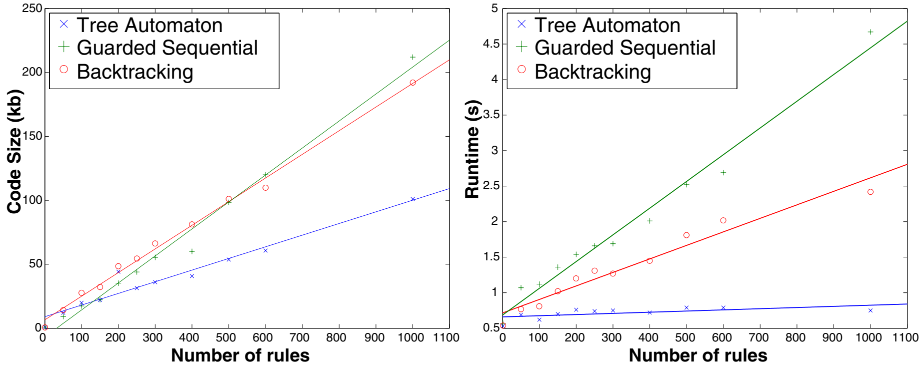
As we can see in Fig. 7.1, the runtime of the tree automaton approach stays more or less constant as the number of patterns increases. The compilation times and size requirements for the tree automaton approach grow linearly. The reason that the code size only grows linearly with the number of patterns is that in the rule-set we use there are a lot of interfering actions. This results in a lot of pruning, which in turn helps limit the size.

It is clear that the guarded sequential automaton approach suffers from the fact that both runtime and code size depends quite heavily on the number of binary

Table 5. *Impact of number of rules*

	Tree Automaton			Guarded Sequential			Backtracking		
Rules[a]	Size[b]	CompT[c]	RunT[d]	Size	CompT	RunT	Size	CompT	RunT
1	0.7	0.1	0.53	0.9	0.3	0.57	0.7	0.1	0.54
50	12.1	2.4	0.69	9.1	1.8	1.07	14.1	3.5	0.77
100	19.9	3.9	0.62	17.4	2.9	1.12	27.8	7.9	0.81
150	22.1	4.7	0.70	22.3	3.8	1.36	32.3	10.2	1.02
200	44.2	6.5	0.76	35.2	6.5	1.54	48.6	17.1	1.20
250	31.4	7.7	0.74	44.0	8.4	1.66	54.6	21.3	1.31
300	35.3	8.9	0.75	49.8	9.5	1.68	64.4	26.1	1.27
400	40.8	10.4	0.72	60.1	11.3	2.01	81.2	33.0	1.45
500	53.7	15.0	0.79	98.4	18.4	2.52	101.0	55.0	1.81
600	60.7	18.2	0.79	120.0	26.2	2.69	110.0	75.2	2.02
1000	101.0	35.3	0.75	212.0	55.5	4.67	192.0	159.0	2.42

[a] Number of rules.
[b] Size of the generated AMD64 native code (in KBytes).
[c] Compilation times (in secs).
[d] Run times (in secs).



(a) Code sizes

(b) Runtimes

Fig. 13. Graphs corresponding to data in Table 5.

patterns. The results for BEAM's backtracking approach shows that the performance of this approach also deteriorates as the number of patterns grows.

For the acl1 rule set specification of ClassBench, the tree automaton approach generates the smallest and fastest code. We were somewhat surprised by the code size result, and indeed it is a fluke, but we show the results using acl1 nevertheless to highlight the fact that the tree automaton approach can in practice actually be more economical in space than the guarded sequential automaton when the amount of sharing is considerable. In the case of acl1, pruning based on interference of match tests (section 5.6.2), which is unique to the tree automaton approach, is very effective. For other filter sets the picture is however different. Table 6 shows the resulting code sizes for nine different rule set specifications distributed with the

Table 6. *Code sizes for different filter sets (in Kbytes)*

Filter Set	Tree Automaton	Guarded Sequential	Backtracking
acl1	35.3	49.8	64.4
acl2	202.0	60.7	62.5
acl3	241.0	87.9	86.1
acl4	194.0	92.3	86.2
acl5	40.3	53.7	67.9
fw1	335.0	39.1	60.1
fw3	385.0	42.7	57.9
ipc1	251.0	72.6	74.9
ipc2	73.8	21.9	32.9
Average size	195.0	57.9	65.9
Std. deviation	125.0	23.1	16.2

ClassBench framework. As expected, the guarded sequential automaton approach generates the smallest code on average. The tree automaton approach generates the smallest code for two of the benchmark sets, but for the other sets it often generates significantly larger code than both the guarded sequential and the backtracking automaton approach. On the other hand, the tree automaton approach is clearly the fastest. For this reason, the remainder of the performance section concentrates on the tree automaton approach.

7.2 Impact of pruning heuristics and optimizations

To evaluate the impact of pruning heuristics and optimizations, we selected as benchmark programs three different (parts of) actual protocol applications that perform binary pattern matching. The `BER-decode` matching code is quite complicated and contains 14 different patterns and 10 distinct read actions. `BS-extract` contains just four 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 seven 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 Section 5.4. Both size and two time-related aspects of the heuristics are reported in Table 7: the average and maximum height of the DAG.

In Table 7, 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. The time properties of these automata are, however, 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

Table 7. *Impact of heuristics and optimizations*

	BER-decode			BS-extract			PPP-config		
Heuristic	Size[a]	AvgH[b]	MaxH[c]	Size	AvgH	MaxH	Size	AvgH	MaxH
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

[a] Number of nodes in the decision tree automaton when converted into a DAG.
[b] Average height (length of paths from a start node to a leaf node) of the DAG.
[c] Maximum height (length of paths from a start node to a leaf node) of the DAG.

Table 8. *Comparison between programs manipulating binary data written in C and in* Erlang

Program written in	Time
C returning its result as a binary	2.22
Erlang using binary pattern matching	2.58
C returning its result as an Erlang term	4.11
Erlang processing the data in the binary represented using a list of integers	41.06

for time is our current priority, we find the Pruning heuristic to be the most suitable choice. We are currently using it as default.

7.3 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 were fortunate to find 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 supposed to be a stand alone program (first row of Table 8) and one which is supposed to be 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 is included 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 in the form of a list of integers (a representation which could be a reasonable choice if a binary term were not available in the language).

As seen in Table 8, showing times in secs, 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. (Running with a large initial heap size mostly affects the last two rows of Table 8, as binaries above a certain size are stored off-heap in Erlang/OTP and collected via reference counting; see Gustafsson & Sagonas (2002) for more information.)

8 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 proposed by Augustsson (1985) (see also the description by Wadler (1987)) 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 languages such as in the Objective-Caml and Haskell compilers. Recently, Le Fessant & Maranget (2001), in the context of the Objective-Caml compiler, 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 the rules for taking advantage of match test interference (section 5.6.2).

Deterministic tree automata approaches have been proposed before, e.g. by Baudinet & MacQueen (1985) or by Sekar *et al.* (1995). Such tree-based approaches guarantee that no constructor symbol is inspected twice at runtime, 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, heuristics should be employed to find near-optimal trees. An early work on the subject of appropriate such heuristics is that of Baudinet & MacQueen (1985). In the same spirit, Sekar *et al.* (1995) also suggest 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 and construct automata which are minimal under certain criteria. Finally, Scott & Ramsey (2000) also examine several different pattern matching compilation heuristics (including those of Baudinet & MacQueen (1985) and Sekar *et al.* (1995)) and measure their effects on different benchmarks. However, all these works differ from ours in that they heavily rely on being able to do a constructor-based decomposition of patterns, and to inspect terms in positions which are known statically.

Wallace & Runciman (1998) introduced an API for a bit stream data structure for Haskell by exploiting its foreign language interface. 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 that of Wallace and Runciman as the lazy setting of their work might restrict the traversal order of tests.

Several packet filtering frameworks have been developed by the networking community. Some of them, e.g. PathFinder (Bailey *et al.*, 1994), DPF (Engler & Kaashoek, 1996) and BPF+ (Begel *et al.*, 1999), use the backtracking automaton approach to pattern matching to filter packets. To achieve better performance common prefixes are collapsed in Bailey *et al.* (1994) and Engler & Kaashoek (1996). In contrast, the BPF+ framework employs low level optimizations such as redundant predicate elimination to produce efficient pattern matching code. Redundant predicate elimination achieves many of the same goals as the pruning actions that we perform (section 5.3), incorporates the read factoring optimization of Section 5.6.3, but also implements some more aggressive optimizations (e.g. partial redundancy elimination) and allows for more types of tests than the ones in our framework.

Lakshman & Stiliadis (1998) describe a method for packet classification which is to some extent similar to the guarded sequential automaton approach of Section 6. Their method uses boolean variables to decide how a packet is classified in the same way that our method uses boolean variables to decide which pattern a packet matches. In contrast to guarded sequential automata however, their method does not deal with issues of safety of performing actions; this happens in a preprocessing step. This in turn means that the boolean variables do not need to guard actions and that their method does not need to guarantee that actions are performed in a certain order. Since guarded sequential automata guarantee the safety of all actions (e.g. that read actions access data within bounds), their construction is constrained by a (partial) order in which actions must be performed.

There are also packet classification algorithms which are similar in some respects to the tree automaton approach described in this article. Notable among them are HiCuts (Gupta & McKeown, 2000) and Tuple Space Search (Srinivasan *et al.*, 1999). Finally, McCann & Chandra (2000) propose an external type system for packet data which allows for type checking of packets and suggest a scheme to use pattern matching based on type refinement to construct efficient packet filters.

9 Concluding remarks

From the examples of section 3 and the performance data in Table 8 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 article are worthwhile additions to the language. Indeed since 2000, when a notation for binary pattern matching was introduced to Erlang, binaries have been heavily used in commercial applications and programmers have often found innovative uses for them.

Our adaptive binary pattern matching compilation scheme is already part of the Erlang/OTP system from Ericsson (since release 10, October 2004), and Erlang programmers have already benefited from it. The ideas we presented are, however, generic. For this reason, we hope that other high-level programming languages, which employ pattern matching, will also benefit from them.

References

Augustsson, L. (1985) Compiling pattern matching. In: Jouannaud, J.-P. (ed.), *Functional Programming Languages and Computer Architecture: LNCS 201*, pp. 368–381. Springer-Verlag.

Baboescu, F. and Varghese, G. (2001) Scalable packet classification. *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 199–210. ACM Press.

Bailey, M. L., Gopal, B., Pagels, M. A., Peterson, L. L. and Sarkar, P. (1994) PathFinder: A pattern-based packet classifier. *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pp. 115–123.

Baudinet, M. and MacQueen, D. (1985) *Tree pattern matching for ML*. Unpublished paper.

Begel, A., McCanne, S. and Graham, S. L. (1999) BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 123–134. ACM Press.

Engler, D. R. and Kaashoek, M. F. (1996) DPF: Fast, flexible message demultiplexing using dynamic code generation. *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 53–59. ACM Press.

Gupta, P. and McKeown, N. (2000) Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, **20**(1), 34–41.

Gupta, P. and McKeown, N. (2001) Algorithms for packet classification. *IEEE Network*, **15**(2), 24–32.

Gustafsson, P. and Sagonas, K. (2002) Native code compilation of Erlang's bit syntax. *Proceedings of ACM SIGPLAN Erlang workshop*, pp. 6–15. ACM Press.

Hyafil, L. and Rivest, R. L. (1976) Constructing optimal binary decision tress is NP-complete. *Infor. Process. Lett.* **5**(1), 15–17.

Lakshman, T. V. and Stiliadis, D. (1998) High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 203–214. ACM Press.

Le Fessant, F. and Maranget, L. (2001) Optimizing pattern matching. *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pp. 26–37. ACM Press.

McCann, P. J. and Chandra, S. (2000) Packet types: Abstract specification of network protocol messages. *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 321–333. ACM Press.

Nyblom, P. (2000) The bit syntax - the released version. *Proceedings of the Sixth International Erlang/OTP User Conference*. Available at http://www.erlang.se/euc/00/.

Scott, K. and Ramsey, N. (2000) *When do match-compilation heuristics matter?* Technical report CS-2000-13, Department of Computer Science, University of Virginia.

Sekar, R. C., Ramesh, R. and Ramakrishnan, I. V. (1995) Adaptive pattern matching. *SIAM J. Comput.* **24**(6), 1207–1234.

Srinivasan, V., Suri, S. and Varghese, G. (1999) Packet classification using tuple space search. *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 135–146. ACM Press.

Taylor, D. E. (2004) *Models, algorithms, & architectures for scalable packet classification.* PhD thesis, Sever Institute of Washington University.

Wadler, P. (1987) Efficient compilation of pattern matching. In: Peyton Jones, S. L. (ed.), *The Implementation of Functional Programming Languages*, pp. 78–103. Prentice-Hall International.

Wallace, M. and Runciman, C. (1998) The bits between the lambdas: Binary data in a lazy functional language. *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pp. 107–117. ACM Press.

Wikström, C. and Rogvall, T. (1999) Protocol programming in Erlang using binaries. *Proceedings of the Fifth International Erlang/OTP User Conference*. Available at http://www.erlang.se/euc/99/.