Applications, Implementation and Performance Evaluation of Bit Stream Programming in Erlang

Per Gustafsson[1] and Konstantinos Sagonas[2]

[1] Uppsala University and Ericsson AB, Sweden
[2] National Technical University of Athens, Greece

Abstract. Writing code that manipulates bit streams is a painful and error-prone programming task, often performed via bit twiddling techniques such as explicit bit shifts and bit masks in programmer-allocated buffers. Still, this kind of programming is necessary in many application areas ranging from decoding streaming media files to implementing network protocols. In this paper we employ high-level constructs from declarative programming, such as pattern matching at the bit level and bit stream comprehensions, and show how a variety of bit stream programming applications can be written in a succinct, less error-prone, and totally memory-safe manner. We also describe how these constructs can be implemented efficiently. The resulting performance is superior to that of other (purely) functional languages and competitive to that of low-level languages such as C.

1 Introduction

Binary data is everywhere. Many applications such as processing network data, encoding and decoding streaming media files, file compression and decompression, cryptography etc. need to process such data. Consequently, programmers often find themselves wanting to write programs that manipulate bit streams. In imperative languages such as C, processing of bit streams typically happens using so called *bit twiddling* techniques that involve combinations of shifts, bitwise operators and explicit masks on programmer-allocated buffers. In general, bit twiddling obfuscates the intention of the programmer, is often error-prone, and leads to code that is unnecessarily verbose, hard to read and modify. Furthermore, bit twiddling code tends to lose the connection with the specification of the data format which is to be processed.

Declarative languages can in principle avoid these shortcomings since they allow for high-level manipulation of data. Unfortunately, the ability to do so comes with a catch. For example, the pattern matching facilities offered by most functional languages are tightly coupled to constructor-based datatypes. As a result, programmers who want to manipulate bit streams have to choose between the lesser of the following two evils: either pay a significant cost in time and space and convert binary data to a symbolic representation, or resort to an imperative style of programming using bit twiddling techniques on byte arrays. In typical applications which require bit stream manipulation, performance considerations are paramount. As a result, in most practical uses, the imperative style of programming wins although there is no fundamental reason for declarative languages to lack constructs for efficient bit stream manipulation.

Since 2001, the functional language Erlang comes with a *byte-oriented* datatype (called *binary*) and with constructs to do pattern matching on a binary [13]. We have been heavily involved in this work and implemented a scheme for native code compilation of binaries and designed efficient algorithms for constructing deterministic pattern matching automata for byte-based binaries [7]. In last year's Erlang workshop we put forward a proposal [6] for lifting the restriction that Erlang binaries are sequences of bytes rather than bits and described the semantics of bit-level pattern matching on a bit-level binary (called *bit stream*). We have subsequently realized this proposal and describe its applications and implementation in this paper.

More specifically, the contributions of this paper are as follows:

- We explain how declarative programming constructs such as pattern matching and comprehensions brought down to the bit level can simplify bit stream programming (Sect. 2) and show how these constructs allow us to obtain compact and elegant solutions to important real-world applications (Sect. 3).
- We describe how these bit-level constructs can be implemented efficiently (Sect. 4).
- Finally, we compare the efficiency and ease of programming of using this approach to writing bit stream applications, with that of using other languages, both functional and imperative (Sect. 5).

2 Bit Stream Programming in Erlang

We show the features and expressive power of bit stream manipulation in Erlang through a series of examples. A more detailed and formal treatment can be found in [6].

2.1 Constructing and matching a bit stream

This first example is very simple. It shows how to construct a bit stream and how such a stream can be deconstructed using bit-level pattern matching.

```
case <<8:4, 63:6>> of
  <<A:7, B/bitstr>> -> {A,B}
end
```

The expression `<<8:4, 63:6>>` evaluates to a ten-bit bit stream were its first four bits are the four low bits of the integer 8 and its last six bits are the six low bits of the integer 63. This creates the bit stream `<<1000111111>>`. For succinctness, we will denote such a bit stream as `<<143:8, 3:2>>`, which means that the first eight bits of the bit stream represented as an unsigned integer is 143 and the last two bits are the integer 3.

The `case` statement binds the variable `A`[3] to an integer constructed from the first seven bits in the bit stream, namely 39 (1000111). Because of the explicit type specifier `bitstr` rather than `integer` which is the default, `B` gets bound to the remaining bit stream `<<7:3>>`. As a result, the `case` expression evaluates to `{39,<<7:3>>}`.

Another useful feature of bit streams is the ability to have arithmetic expressions as sizes of bit stream segments. This is shown in the next example.

[3] All variables in Erlang start with a capital letter.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
parse_IP_packet(
  <<Version:4, IHL:4, ToS:8, TotalLength:16,
    Identification:16, Flags:3, FragOffset:13
    TimeToLive:8, Protocol:8, Checksum:16
    SourceAddress:32,
    DestinationAddress:32,
    OptionsAndPadding:((IHL-5)*32)/bitstr,
    Data/bitstr>>) when Version =:= 4 ->
  ...
```

Fig. 1. Internet Protocol datagram header (from RFC 791) and parsing of an IPv4 packet in Erlang

2.2 Parsing IP packets

In RFC 791 [14] the IP header is exemplified with the diagram shown in the left part of Figure 1. Note the close resemblance between this representation and the bit stream pattern shown in the right part of the figure which parses an IPv4 packet header.

For the most part, this is similar to the previous example except that this pattern is used in a function head rather than a `case` statement. Note also that the pattern expresses the meaning of the `IHL` field, which contains the IP header length in 32-bit words. Since the non-optional part of the IP header consists of five 32-bit words, the options and padding will take up `(IHL-5)*32` bits. This is expressed by using an arithmetic expression as the size of a segment. Because this arithmetic expression can refer to variables bound earlier in the binary pattern, as in this example, the matching has to respect the corresponding left-to-right ordering constraints between segments.[4]

2.3 Iterating and filtering a bit stream

Consider a variation of the `drop_third` program introduced in [17] that requires inspecting bits besides counting them. The task is to drop from a bit stream of size exactly divisible by three all 3-bit chunks that begin with a zero. Using pattern matching on bit streams this task can be performed with the program in Figure 2. The solution is both natural and straightforward. The first clause describes what should happen if the first bit in a 3-bit chunk is one: we keep that chunk and add it to the resulting stream. The second clause handles the case where the first bit is a zero: we discard that 3-bit chunk. Finally the last clause handles the case where there are no more chunks: we return the empty bit stream.

```
drop_0XX(<<1:1, X:2, Rest/bitstr>>) ->
  <<1:1, X:2, drop_0XX(Rest)>>;
drop_0XX(<<0:1, _:2, Rest/bitstr>>) ->
  drop_0XX(Rest);
drop_0XX(<<>>) ->
  <<>>.
```

Fig. 2. `drop_0XX` using bit stream pattern matching

Contrast this with a program written in a language that does not support manipulation of bit streams at the bit level very well such as C or Java. The programmer would

[4] Arithmetic expressions as sizes of segments are not allowed in Erlang/OTP R11B-1 (or prior). Instead, size expressions can only be variables or constants.

have to keep track of which bits to extract from the current byte of the incoming bit stream, use bit masks and shifts to extract each triple, and calculate how much padding is needed in the output stream. Being able to express pattern matching at the bit level, Erlang programmers are allowed to write declarative specifications of their intentions without having to worry about low-level details such as padding.

2.4 Inverting a bit stream using a comprehension

Another way to write code which iterates over a bit stream is to use a bit stream comprehension [6]. This is a construct analogous to a list comprehension [18], which in turn is an expression that is syntactic sugar for the combination of `map`, `filter` and `concat` on lists. For a simple example use of a bit stream comprehension consider the task of inverting all bits in a bit stream. The `bsnot` function below performs this task.

```
bsnot(BitStr) ->
  << bnot(X):1 || <<X:1>> <= BitStr >>.
```

The meaning of this comprehension is: iterate through each bit in the bit stream, invert it using the built-in `bnot` operator, and put it into the resulting bit stream.

2.5 Iterating and filtering a bit stream using comprehensions

For a slightly more involved example consider the `drop_0XX` function of Section 2.3. Using bit stream comprehensions, `drop_0XX` would be written more succinctly as:

```
drop_0XX(BitStr) ->
  << <<1:1,X:2>> || <<1:1,X:2>> <= BitStr >>.
```

This comprehension works as follows. If the first three bits of the bit stream match the pattern `<<1:1,X:2>>` then place those bits in the resulting stream; otherwise drop these bits. Repeat until no bits remain in the bit stream. That is the pattern works as both a filter and a generator. To make this more explicit we can write a `drop_0XX` function which is equivalent with the previous one using an explicit filter in the following manner:[5]

```
drop_0XX(BitStr) ->
  << X:3 || <<X:3>> <= BitStr, 2#100 =< X >>.
```

In bit stream comprehensions, sometimes more complicated, perhaps user-defined, filtering is needed. In the following example, we are given a string represented as a bit stream and want to extract all non-digit characters from this string and store each of the digits in four bits:[6]

```
compact_digits(String) ->
  << (X-$0):4 || <<X:8>> <= String, is_digit(X) >>.

is_digit(X) when $0 =< X, X =< $9 -> true;
is_digit(_) -> false.
```

[5] In Erlang, `2#100` represents the number four in base two.

[6] In Erlang, `'$'` is an operator which given a character returns the ASCII value of that character.

3 Applications

3.1 UU-encoding

UU-encoding is an old binary-to-text encoding scheme where groups of three binary bytes are encoded in four characters. This is done by dividing the three binary bytes into four groups of six bits. Then 32 is added to each six bit group which turns them into characters. The cores of these encoding and decoding scheme essentially become one-liners using Erlang's bit stream programming facilities.

```
uuencode(BitStr) ->
  << (X+32):8 || <<X:6>> <= BitStr >>.

uudecode(Text) ->
  << (X-32):6 || <<X:8>> <= Text >>.
```

3.2 yEnc

The yEnc format is a newer encoding of binary files than UU-encoding where bytes which cannot be safely transmitted in text mode are escaped. Each byte in the original stream is encoded by adding 42 to it using 8-bit arithmetic. If the result is a critical character (i.e., NULL, TAB [ASCII 9], LF [ASCII 10], CR [ASCII 13] which are hard to transmit over some networks or '=' [ASCII 61] which is used as an escape character), the character is encoded using two bytes: the first byte is '=' [ASCII 61] and the second byte is the critical value plus 64. To encode a binary file in the yEnc format [8], we can use the bit stream comprehension in the following program:

```
yenc(Bin) ->
  << yenc_byte(Byte) || <<Byte:8>> <= Bin >>.

yenc_byte(Byte) ->
  Enc = (Byte+42 rem 256),
  case is_critical(Enc) of
    true  -> <<61:8, (Enc+64):8>>;
    false -> <<Enc:8>>
  end.
```

3.3 μ-law

Audio files are nowadays transmitted over the network using a variety of formats. One such format, designed to be space efficient, is μ-law compressed files [10]. Such files are compressed to half the size of the original audio as each 16-bit sample is translated into an 8-bit representation.

μ-law encoding The encoding method is non-trivial but still quite simple. First the `Sound` sample is transformed from 2's complement form to a `Biased` sign magnitude form where the magnitude is an integer in the range [132..32767]. This can be done easily with the bit stream comprehension:

```
<< to_sign_magn(Sample) || <<Sample:16/integer-signed>> <= Sound >>
```

which simply takes each 16-bit sample in 2's complement form. This is achieved by using the signed specifier in the pattern. The to_sign_magn function is then applied to this value. This function is defined as follows:

```
to_sign_magn(Sample) ->
  <<sign(Sample):1, (min(abs(Sample), 32635)+132):15>>.
```

i.e., it transforms the sample from 2's complement form into sign magnitude form and increases the magnitude with 132.

In the next step, this representation is translated to an 8-bit representation where the first bit represents the sign, the next three bits represent the position of the first 1 in the magnitude, and the last four bits represent the values of the four bits following the leading 1. This can also be done with a comprehension of the form:

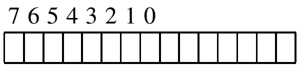
```
<< to_byte(S,M) || <<S:1,M:15/bitstr>> <= Biased >>
```

In this case, S contains the sign bit and M is a bit stream consisting of 15 bits representing the magnitude of the sample. These are used as arguments to the to_byte function which is defined as follows:

```
to_byte(Sign, Magn) -> to_byte(Sign, Magn, 7).

to_byte(Sign, <<1:1, Mantissa:4, _/bitstr>>, N) ->
  <<Sign:1, N:3, Mantissa:4>>;
to_byte(Sign, <<0:1, Rest/bitstr>>, N) ->
  to_byte(Sign, Rest, N-1).
```

This function searches for the position of the first 1 in the Magn bit stream. Since the range of the magnitude is 132–32676 there will be at least one 1 in the first 8 bits and recursion will stop. The position of the first 1 is therefore coded in the following way:



Thus, if the third bit contains the first 1, its position is 5. The following four bits are called the mantissa. In the byte created by the to_byte function the first bit contains the sign, the following three bits contain the position, and the last four bits contain the mantissa.

Finally, we take the 1's complement of this value using the bsnot operator of Section 2.4. The complete code for μ-law encoding is shown in the appendix.

μ-law decoding To decode these values we start by taking their 1's complement. We then translate the bytes to sign magnitude form again with this comprehension:

```
Biased = << to_short(Sign, Exp, Mantissa) ||
            <<Sign:1,Exp:3,Mantissa:4>> <= Encoded >>
```

where the to_short function is defined in the following way:

```
to_short(Sign, Exp, Mantissa) ->
  <<Sign:1, 1:(8-Exp), Mantissa:4, 1:1, 0:(2+Exp)>>.
```

That is, put the Sign bit first, then put the leading one in the correct place followed by the mantissa and an additional 1 and fill the remaining bits with zeroes.

Finally, we must translate the sign magnitude representation into 2's complement representation and remove the bias. This is done with the comprehension:

```
<< unbias(Sign,Magn) || <<Sign:1,Magn:15>> <= Biased >>
```

where the function unbias is defined as follows:

```
unbias(0, Magn) -> <<(Magn - 132):16>>;
unbias(1, Magn) -> <<(132 - Magn):16>>.
```

3.4 PNG

The Portable Network Graphics (PNG) file format [16, 11] is a rather recent format for picture files intended to replace the widely-used but patent-based GIF format. The structure of the PNG format is quite simple. It consists of an initial signature and then a series of chunks. Each of the chunks consists of a length field, a type field, the chunk data, and a checksum. A certain type of chunk contains the raw compressed data whereas the rest of the chunks contains meta data. Assuming that the PNG variable is bound to a bit stream where we have removed the signature from the original file, we can recreate the raw data in order to decompress it using the following bit stream comprehension.

```
<< RawData || <<Length:32, 73:8,68:8,65:8,84:8,
                RawData:(Length*8)/bitstr, _Crc:32>> <= PNG >>
```

The sequence of numbers 73,68,65,84 is the content of the type field for the chunk containing raw data. This means that only the chunks that contain raw data match the generator pattern and only the data from those chunks makes up the resulting bit stream. We can then decompress this data and use the uncompressed data and the chunks containing meta data to generate the picture.

3.5 Huffman

Huffman encoding is a variable length encoding of characters. The mapping between the variable length codes and the static codes is described by a *Huffman tree*. This tree is a binary tree where the leaves are static codes. The mapping from the dynamic length codes to the static codes is encoded in the path from the root to a leaf. For example, if a leaf contains the static code 32 and is reached from the root by taking the left branch, then the right branch and finally the left branch, this means that 010 maps to 32.

To decode a Huffman encoded bit stream we can use Program 1. The main decoding function has four clauses. The first is taken if we have reached a leaf in the Huffman tree. If this is the case we add the value in that leaf to the output and recurse. The second clause is taken if we are at a branch and the value of the next bit is zero. In that case we take the left branch. The third clause is taken if the next bit was one and in that case we choose the right branch. The fourth and final clause is taken when there are no more bits left to decode which means that we are done.

Program 1 Function for decoding a Huffman encoded bit stream

```
huffman_decode(BitStr, Tree) ->
  huffman_decode(BitStr, Tree, Tree).

huffman_decode(Rest, Char, Tree) when is_char(Char) ->
  [Char | huffman_decode(Rest, Tree, Tree)];
huffman_decode(<<0:1,Rest/bitstr>>, {Left,_}, Tree) ->
  huffman_decode(Rest, Left, Tree);
huffman_decode(<<1:1,Rest/bitstr>>, {_,Right}, Tree) ->
  huffman_decode(Rest, Right, Tree);
huffman_decode(<<>>, _, _) ->
  [].
```

4 Implementation

Having seen constructs and typical applications of bit stream manipulation, let us now see how we efficiently implement these constructs.

4.1 Internal representation of bit streams

We have chosen an internal representation of bit streams which has the property that the space overhead of storing each stream is constant, independent of the size of the stream. The representation uses two different structures: a *base stream* and a *sub-stream.* The base stream contains a header, a size field expressing the size of the bit stream in bits, and an array of data which contains the actual bit sequence. For a bit stream with bit size `n`, the bit sequence starts with the first bit in the data array and ends at the `n`-th bit in the array. The sub-stream structure contains a header field, a size field, an offset field, and a pointer to a base stream. Let us denote the content of the size field by `n`, the content of the offset field by `o` and the base stream that the sub-stream is pointing to by `BS`. Then the bit sequence that the sub-stream represents starts with `o`-th bit of the data array of the base stream `BS` and ends with the (`o+n-1`)-th bit of the data array of `BS`.

Figure 3 shows the representation of a base stream and two sub-streams. In our implementation, the header, size and offset fields are all word-sized even though they look smaller in the figure. The header field stores the size in words of the structure and a runtime tag which identifies the object as a base stream (or sub-stream). In the figure, `A`, `B`, and `C` are all variables bound to binaries. `A` is bound to the base stream `<<47:8,47:8,101:8,1:7>>`, `B` is bound to the sub-stream `<<25:5>>` marked with a black border in the figure and `C` is bound to the sub-stream `<<47:8,101:8>>` which is marked with a grey background in the figure.

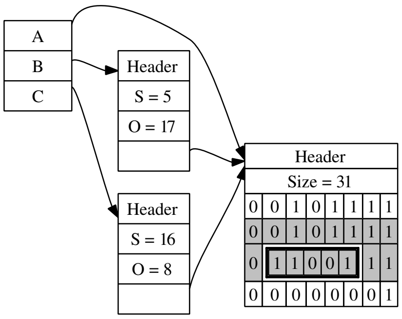


Fig. 3. Internal representation of bit streams

4.2 Implementation of bit stream construction

Bit stream construction is aided by two low-level auxiliary functions:

put_integer() which given a pointer, an offset in bits, a size in bits, and an integer (a fixnum or a bignum) and writes size bits of the integer starting at offset bits from the pointer, and

put_bitstr() which given a pointer, an offset in bits, a size in bits, and a bit stream writes the first size bits of the bit stream starting at offset bits from the pointer.

A bit stream construction expression of the form $\texttt{<<ve}_1\texttt{:se}_1\texttt{/t}_1\texttt{,...,ve}_n\texttt{:se}_n\texttt{/t}_n\texttt{>>}$ is translated using these functions as follows. We start by evaluating all the value and size expressions and end up with an expression of the form $\texttt{<<v}_1\texttt{:s}_1\texttt{/t}_1\texttt{,...,v}_n\texttt{:s}_n\texttt{/t}_n\texttt{>>}$ where all the $\texttt{v}_i$:s are values and all the $\texttt{s}_i$:s are non-negative integers. If any $\texttt{s}_i$ is a negative value, a run-time exception is raised.

Then, we perform the following operations:

1. Calculate the resulting size of the bit stream as $\sum_{i=1}^{n} s_i$.
2. Allocate a base stream with a large enough data array to hold all the bits of the bit stream, initialize data_ptr to a pointer to the beginning of the data array and set offset to 0.
3. For each segment, do the following:
   (a) If $\texttt{t}_i$ is `integer` we call put_integer(data_ptr, offset, s_i, v_i)
   (b) If $\texttt{t}_i$ is `bitstr` we call put_bitstr(data_ptr, offset, s_i, v_i)
   (c) Set offset to offset+s_i
4. After all segments are processed, return the base stream.

4.3 Implementation of bit stream pattern matching

We only describe the case of matching a bit stream against a single binary pattern. For a thorough treatment of how to efficiently match a (byte-aligned) binary against many patterns simultaneously refer to our prior work [7] which describes effective algorithms for constructing deterministic binary pattern matching automata.

The matching is aided by two low-level auxiliary functions:

get_integer() which returns an integer given a pointer to some data, an offset in bits into that data, and the number of bits that should be used to create the integer, and

get_bitstr() which creates a sub-stream from an offset, a size and a pointer to a base-binary.

To match $\texttt{<<X}_1\texttt{:e}_1\texttt{/t}_1\texttt{,...,X}_n\texttt{:e}_n\texttt{/t}_n\texttt{>>}$ against a bit stream `BitStr` we perform the matching in the manner described below.

1. Create a matching state from `BitStr`. The state contains the following information:
   data_ptr a pointer to the data
   offset the present offset into the data
   end the offset of the last bit in the stream
   orig_ptr a pointer to the base stream which contains the data

2. For each segment, perform the following tasks:
   (a) Evaluate $\mathtt{e}_i$, the size expression of the first segment to the integer $\mathtt{s}_i$.
   (b) Check whether offset+$\mathtt{s}_i \leq$ end, or else the matching fails.
   (c) If $\mathtt{t}_i$ is `integer` then $\mathtt{X}_i$ = get_integer(data_ptr, offset, $\mathtt{s}_i$)
   (d) If $\mathtt{t}_i$ is `bitstr` then $\mathtt{X}_i$ = get_bitstr(offset, $\mathtt{s}_i$, orig_ptr)
   (e) Set offset to offset+$\mathtt{s}_i$
3. Check whether offset == end. If so, the matching succeeds, otherwise it fails.

A tail segment (i.e., a last segment of the form $\mathtt{X}_n$`/bitstr`) is handled specially: we bind $\mathtt{X}_n$ to get_bitstr(offset, end, orig_ptr) and set the value of offset to end.

Also, note that we described the case where all segments are of the form $\mathtt{X}_i\mathtt{:e}_i\mathtt{/t}_i$ where $\mathtt{X}_i$ is a variable. If some $\mathtt{X}_i$ is not a variable but has a value $\mathtt{v}_i$ we simply add an equality test that checks that $\mathtt{v}_i$ is equal to the value returned from either get_integer or get_bitstr. If not equal, the matching fails. Otherwise the matching continues with the next segment.

4.4 Efficient abstractions and alternatives

With the contiguous internal representation of Sect. 4.1 bit stream pattern matching is fast but building bit streams piece by piece is expensive. Still, on top of our representation we can build two efficient abstractions, *segmented bit streams* and *buffers*.

Segmented bit streams A segmented bit stream consists of a list of (possibly segmented) streams and represents the stream that is formed if the streams in the list are concatenated. Thus, a segmented bit stream is a (deep) list of bit streams. This abstraction makes it easy and cheap to concatenate a new bit stream to an existing segmented bit stream: all we need to do is to put it first in the list. Then, to efficiently turn a segmented bit stream into a regular contiguous bit stream we introduce a built-in called list_to_bitstr which simply transforms a (deep) list of bit streams into a single, contiguous one. This way, constructing a bit stream of size n piecemeal from some other streams can be done in $O(n)$ as opposed to $O(n^2)$ if segmented bit streams are not used. However, note that in the worst case (when each element in the list is a one-bit stream), the segmented bit streams abstraction has a significant space overhead.

Buffers The idea of the buffer abstraction is taken from the Lua programming language [9]. A buffer is basically a list of bit streams with the following invariant: each bit stream in the list is strictly smaller than the next bit stream in the list. Note that, since the representation is a list of bit streams, the list_to_bitstr built-in can then be used to turn a buffer into a contiguous bit stream. However, since we need to maintain the invariant that bit streams in the list are increasing in size, sometimes we need to concatenate bit streams directly when adding streams to the buffer. This makes construction of a buffer more expensive than constructing a segmented bit stream, but the invariant keeps the space overhead lower for a buffer than for a segmented bit stream, since the maximal length of the list is $O(\sqrt{n})$ if the total number of bits is n.

Currently, neither *buffers* nor *segmented bit streams* have any support on the language level. This means that e.g. to use bit stream pattern matching on a buffer, the buffer must first be explicitly converted to a contiguous representation using list_to_bitstr.

4.5 Implementation of bit stream comprehensions

The implementation of bit stream comprehensions requires considering the implications of the chosen underlying representation. If we choose to implement bit stream comprehension naïvely, constructing a new bit stream in each iteration the cost of the comprehension would be quadratic in the number of iterations.

Naturally we can do better than this. One possible choice is to use segmented bit streams, i.e. build a list of bit streams and then use the list_to_bitstr built-in to convert the list into a bit stream. Another possibility is to collect all of the bit streams in a list accumulator and at the same time calculate the sum of the sizes of the streams in the list. In this way we find out the size of the resulting bit streams and create a list whose elements are the streams in reverse order. We can then allocate a large enough base stream and copy the bit streams in the list into the data array of that base stream.

Though both these solutions have linear complexity, we can decrease the constant factors significantly whenever it is possible to compute an upper bound on the size of the resulting bit stream. In these cases we allocate a base stream in advance and write the results to the base stream as the bit stream comprehension is evaluated.

When is it possible to compute an upper bound on the resulting bit stream? Let us consider the case when we only have one generator, which is by far the most common situation. In such a case, the bit stream comprehension looks as follows:

$$\texttt{<<}\ e\texttt{:}se/t\ \texttt{||}\ \texttt{<<}e_1\texttt{:}se_1/t_1,\ldots,e_n\texttt{:}se_n/t_n\texttt{>>}\ \texttt{<= BitStr,}e_f\ \texttt{>>}$$

If all of the size expressions $(se, se_1, \ldots, se_n)$ can be evaluated before the bit stream comprehension starts being evaluated, then we can calculate how many bits of the input bit stream are consumed in each iteration $(\sum_{i=1}^{n} se_i)$ and how many bits might be produced in each iteration (se). That is, if BitStr has size m the maximal number of bits in the resulting binary is: $(m \times se)/\sum_{i=1}^{n} se_i$.

On the other hand, in some cases it is impossible to calculate a tight upper bound on the size of the resulting binary. One example is this comprehension:

```
<< 42:N || <<S:8,N:(S*S)>> <= BitStr >>
```

Luckily, such comprehensions are rather rare in practice. Thus, in our implementation we chose to stick to a simple implementation of bit stream comprehensions, namely that which uses segmented binaries flattened by a call to list_to_bitstr for such uncommon cases. For cases when a tight upper bound can be calculated we use the method which preallocates a base stream of suitable size.

5 Performance

From Section 3 it should be clear that bit-level binaries and comprehensions allow for flexible manipulation of bit streams. Still, these constructs are to be used in applications where speed of processing is a prime consideration. Thus, it is imperative that the performance of the underlying implementation is competitive with both imperative languages using bit shifts and bit masks on byte arrays and with other high-performance functional languages using bit or byte arrays for representing bit streams.

drop_0XX This is the program from Section 2.3. It takes a bit stream and removes all 3-bit chunks that start with a 0. In the benchmark, the size of the input stream is about 28.5 million bits; the size of the resulting bit stream is about 8 million bits. We perform 10 iterations.

five11 Implements the IS-683 PRL protocol. Reads a file whose first 16 bits represent an integer that describes how many PRL packets the file contains. Each packet starts with a 5-bit integer describing how many channels the packet contains and is followed by that number of 11-bit channel descriptors. The output is a list of channel descriptors for each packet. The input data consists of 496 different packets (16 of each possible size) and is decoded 10,000 times.

huffman The input is a file containing the huffman tree and a message encoded using this tree. The benchmark recreates the original message. The size of the encoded file is 747,647 bytes and the decoded file consists of 3,568,560 bytes. The file is decoded 10 times.

uudecode This benchmark decodes a file that has been uuencoded. The size of the encoded input file is 747,647 bytes and the size of the decoded output file is 542,623 bytes. The file is decoded 100 times.

uuencode This benchmark uuencodes a file. The input file consists of 542,623 bytes and the encoded output consist of 747,647 bytes. The file is encoded 100 times.

Fig. 4. Description of the benchmarks

Notice however that bit streams in Erlang are immutable data structures. The language provides no support for destructive updates. Also, notice that memory management for bit streams is automatic and a responsibility of the underlying runtime system, not of the programmer. Thus comparing the performance of functional vs. imperative languages in applications which manipulate bit streams has a bit of an “apples and oranges” flavor, especially since different styles of programming are often employed.

Still, this performance comparison is interesting. We will base it on the programs described in Figure 4 which spend the bulk of their work in bit stream manipulation.

We have implemented these benchmarks in three different functional languages, namely Erlang with all the extensions described in this paper, Haskell and O’Caml. In addition, we wrote C and Java versions of the first three benchmarks and found publicly available **uudecode** and **uuencode** C programs on the net which we converted to appropriate benchmarks and translated to Java. Our intention was to eliminate any traces of possible favoritism for some language and any inefficiencies due to our programming skills. So, we requested the help of Haskell and O’Caml experts to perform any efficiency improvements they saw fit, provided that the programs remain functional: i.e., use no mutation in the part of the program for which measurements are taken. On the other hand, the imperative languages are free to—and indeed do—use destructive assignments on all benchmarks.

The compilers that we used are the Glasgow Haskell Compiler version 6.4.1, the O’Caml 3.09.1 native code compiler, and GCC 3.4.2 for C and Java (`gcj`). For Erlang we used the HiPE native code compiler in the pre-release of Erlang/OTP R11B-2. The machine we used is a 2.4 GHz Pentium 4 with 1 GB of memory running Fedora Core 3.

5.1 Runtime performance

Figure 5 shows performance results. We can see that Erlang enhanced with the constructs described in this paper is competitive in speed with other state-of-the-art func-

Runtimes (in secs)

	Functional			Imperative	
Benchmark	Erlang	Haskell	O'Caml	C	Java
drop_0XX	2.09	5.85	2.25	0.96	1.99
five11	4.97	8.65	7.69	9.79	18.41
huffman	2.29	7.38	10.81	0.97	1.75
uudecode	3.21	6.04	2.65	0.86	0.97
uuencode	2.85	7.77	2.82	1.04	0.98

Lines of code

	Functional			Imperative	
Benchmark	Erlang	Haskell	O'Caml	C	Java
drop_0XX	2	47	45	26	47
five11	9	38	23	64	78
huffman	14	30	54	67	81
uudecode	20	91	65	43	57
uuencode	25	70	70	54	64

Fig. 5. Time performance and succinctness of programming in different languages

tional languages in programs that manipulate bit streams. This is not due to Erlang's overall performance compared with Haskell and O'Caml. Instead, it is due to having these constructs in the language and having the compiler generate reasonably efficient code for them. Also, at least for these programs, the performance of the functional way of manipulating low-level representations is not so far away from that obtained using C with destructive assignment and programmer-controlled memory management.

Some runtime numbers stick out and require explanation. The bad performance of O'Caml on **huffman** is due to extensive garbage collection; the program spends more than half of its time doing GC. Also, the bad performance of imperative languages on **five11** is partly due to the nature of the task, which is not tailored to accessing bits in a multiple-of-eight fashion, and partly due to calling individual `malloc`:s and `free`:s (in C) for each channel description rather than allocating a big memory area once and partitioning it to each channel using programmer-controlled pointer bumping.

5.2 Succinctness and ease of programming

Performance is only part of the story. Ease of programming is equally important. It is very difficult to quantify this dimension, but the lines of code required to perform these tasks in different languages provide some rough estimate. As seen in Figure 5, the Erlang solutions are 2–20 times more compact than solutions in other functional languages. Once again, this is not due to the functional core part of Erlang; it is due to the ability to manipulate bit streams declaratively.

We have used the following rules when counting line numbers:

- We only counted lines directly involved in performing the tasks required by the benchmarks, not lines needed for I/O or for measuring execution times.
- We did not count blank lines, comments, type specifications of functions, strictness annotations, or lines containing only one keyword.
- No line was allowed to be wider than 80 characters.

We have made all these benchmark programs publicly available and annotated their source code with line numbers to see exactly which lines we count in the different benchmarks. Their annotated source code is at http://user.it.uu.se/∼pergu/bitbench. Input data for running these programs, further information, as well as a pre-release of the Erlang/OTP system we have used are also accessible from the same site.

6 Related Work

Currently, very few general purpose languages provide constructs for direct manipulation of binary data down at the bit level, let alone efficient ones. Bit streams are typically represented as character arrays and their bit-level manipulation is performed by the programmer using explicit bit shifts and bit masks. Doing so is both exacting and error-prone. But since this kind of programming is commonplace in domains such as cryptography, data communication and multimedia programming, a plethora of domain-specific languages targeting these areas come with some ability to manipulate bit streams.

Cryptol [12] and SLED [15] are domain specific languages in the field of cryptology and machine language manipulation, respectively. They both allow bit-level pattern matching, but the size of the fields in the patterns are fixed at compile time.

Solar-Lezama et al. have proposed BitStream, a language for manipulating binaries in the coding and cryptography area [17]. The dataflow programming model used in BitStream is radically different from ours, as is the methodology to achieve both correct and efficient programs which requires the programmer to first write a simple reference implementation and then sketch a more efficient implementation which is rejected by the compiler if it is not equivalent to the reference implementation. For some applications, BitStream achieves good performance, on par with hand-optimized C programs.

In the area of data communication Chandra and McCann [2] have proposed a type system which can be used to describe how network packets are structured at the bit level. Back has proposed the DataScript [1] language which is both a constraint-based specification language for specifying binary data formats and a scripting language for manipulating such formats. DataScript is based on Java and does not support pattern matching. The PADS [4] language, proposed by Fisher and Gruber, allows description of any ad hoc data format and comes with the ability to automatically generate tools that manipulate such formats. In the context of the PADS project, Fisher, Walker, and Mandelbaum have recently developed a calculus of dependent types [5] which is suitable to use as a semantic foundation for the whole family of data description languages.

The previous examples of related work are all in one way or another domain-specific. Diatchki, Jones and Leslie [3], proposed a language extension for general purpose languages that allows pattern matching on fixed-width bit data types. Their proposal would make it easier to use a high-level functional language similar to Haskell to perform low-level tasks like writing device drivers or implementing operating systems. What distinguishes their work from ours is that 1) they only consider bit data whose representation fits in the registers of a machine while we do not have any such constraint, and 2) that their implementation comes in the form of an interpreter rather than being fully integrated in a general purpose programming language.

7 Concluding Remarks

The treatment of bit-level data is a neglected area in general-purpose programming languages and most declarative languages are no exceptions. This is unfortunate since there are many applications out there craving for language constructs which remove the need for tedious and error-prone bit-twiddling, while still achieving decent performance.

Armed with bit stream comprehensions and the ability to perform pattern matching at the bit level without being hampered by artificial restrictions (e.g., always having to create bit streams whose length is a multiple of eight) we have shown how a variety of important "real-world" bit stream applications can be programmed both succinctly and efficiently. We see very little reason for bit streams not to co-exist with other complex terms such as lists or tuples, or for Erlang to be an exception in providing such support. Perhaps this paper paves the way in this direction.

References

1. G. Back. Datascript — a specification and scripting language for binary data. In *Generative Programming and Component Engineering*, pages 66–77. Springer, Sept. 2002.
2. S. Chandra and P. J. McCann. Packet types. In *Proceedings of the Second ACM SIGPLAN Workshop on Compiler Support for System Software*. ACM Press, May 1999.
3. I. S. Diatchki, M. P. Jones, and R. Leslie. High-level views on low-level representations. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 168–179. ACM Press, Sept. 2005.
4. K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and mplementation*, pages 295–304. ACM Press, June 2005.
5. K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15. ACM Press, Jan. 2006.
6. P. Gustafsson and K. Sagonas. Bit-level binaries and generalized comprehensions in Erlang. In *Proceedings of the Fourth ACM SIGPLAN Erlang Workshop*, pages 1–8. Sept. 2005.
7. P. Gustafsson and K. Sagonas. Efficient manipulation of binary data using pattern matching. *Journal of Functional Programming*, 16(1):35–74, Jan. 2006.
8. J. Helbing. yEnc: Efficient encoding for Usenet and eMail, June 2002. See also www.yenc.org.
9. R. Ierusalimschy. *Programming in Lua*. Lua.org, second edition, Mar. 2006.
10. International Telecommunication Union. *G.711: Pulse code modulation (PCM) of voice frequencies.* Series G: Transmission Sytems and Media, Digital Systems and Networks. Standardization Sector of ITU, Geneva, Switzerland, Nov. 1998.
11. Joint ISO/IEC International Standard and W3C Recommendation. Portable network graphics (PNG) specification, W3C/ISO/IEC version, Nov. 2003.
12. J. R. Lewis and B. Martin. Cryptol: high assurance, retargetable crypto development and validation. In *Proceedings of the IEEE Military Communications Conference*, pages 820–825. IEEE, Oct. 2003.
13. P. Nyblom. The bit syntax - the released version. In *Proceedings of the Sixth International Erlang/OTP User Conference*, Oct. 2000. Available at http://www.erlang.se/euc/00/.
14. J. Postel. RFC 791: Internet Protocol, Sept. 1981. Obsoletes RFC0760. Status: STANDARD.
15. N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *ACM Trans. Prog. Lang. Syst.*, 19(3):492–524, 1997.
16. G. Roelofs. *PNG: The Definite Guide*. O'Reilly and Associates, June 1999. See also www.libpng.org/pub/png/.
17. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioĝlu. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–294. ACM Press, June 2005.
18. P. Wadler. List comprehensions. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 7, pages 127–138. Prentice-Hall, 1987.