

# Functional Manipulation of Bit Streams

Per Gustafsson    Konstantinos Sagonas

Department of Information Technology, Uppsala University, Sweden

{pergu,kostis}@it.uu.se

## Abstract

Writing code that manipulates bit streams is a painful and error-prone programming task, currently 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 show how high-level constructs from functional programming, such as binary pattern matching at the bit level and binary comprehensions, can make bit stream manipulation more succinct, less error-prone, and totally memory-safe. We formally define these language constructs, show a variety of example uses from several different application areas, and describe their implementation in detail. We hold that a binary data structure with the ability to perform pattern matching at the bit level allows (purely) functional languages to significantly expand the range of their practical uses.

## 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 in need to write programs that manipulate binaries. In imperative languages such as C, processing of binary data 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. In addition, bit twiddling code tends to lose the connection with the specification of the data format which is to be processed.

Functional languages can in principle avoid these shortcomings, since they allow for high-level treatment of data using pattern matching techniques. Unfortunately, the ability to do so comes with a catch. The pattern matching facilities offered by most functional languages are tightly coupled to constructor-based datatypes. As a result, functional programmers wanting to manipulate binary data need 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.

There is no fundamental reason for this to continue to be the case. There already exists a functional language with a binary datatype which can be implemented efficiently: Erlang. On the other hand, unfortunately, binaries in Erlang have so far been crippled by various implementation restrictions, most of which are rather ad hoc. Also, Erlang binaries provide only a partial solution to the problem since they have so far been a *byte-oriented* datatype

rather than a *bit-oriented* one. Finally, no formal semantics for pattern matching on binaries has been defined. Thus, despite our recent work on proposing automata for binary pattern matching at the byte level [8], Erlang binaries are not easily adaptable to other functional languages. This paper addresses all these issues.

**Contributions** The contributions of this paper are as follows:

- We extend a language with a functional core with binary values and with the ability to construct such values.
- In this language, which is not tied to Erlang, we introduce and formally define the concept of *bit-level pattern matching* as a natural extension of pattern matching on structured terms.
- We formally define *binary comprehensions*, a derivative of list comprehensions, that operate on bit streams.
- We show how these constructs allow us to obtain compact solutions to interesting “real-world” applications.
- We describe how all these can be implemented efficiently.
- 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.

## 2. A Core Language for Manipulating Binaries

In this section we extend the small strict functional language core shown in Figure 1 with binary values, binary construction, and capabilities for bit-level binary pattern matching. We only present the syntax of the core language, not its operational semantics or any typing rules. This is done for the following reasons. First, its operational semantics is the “standard” one. Similarly, its typing rules are not particularly involved: binaries are just another prime type that does not interact with other prime types. Conversion between binaries and integers<sup>1</sup> is done *only* via (compiler-inserted) built-in conversion functions. Finally, since in the next section we will adapt this core language to Erlang, which is a dynamically typed functional language, we do not want to tie our core language to any particular functional programming language flavor.

### 2.1 The binary datatype

The binary datatype represents a finite sequence of bits. This makes it a versatile container suitable for representing files, network packets, registers, and other low-level constructs.

We represent a binary value as a bit pattern of  $n$  bits prefixed by a '#' (cf. the  $v$  form in Figure 1). All program variables that we use start with an upper case letter. We denote integer values with the letters  $m, n, k$  and  $s$  and binary values of  $n$  bits as  $\beta^n, \xi^n$ , and  $\zeta^n$ . A binary term can also be written as  $\beta_1^{n_1} \oplus \beta_2^{n_2}$ . This is a binary consisting of  $n_1 + n_2$  bits, where the first  $n_1$  bits are the bits of  $\beta_1$  and the following  $n_2$  bits are the bits of  $\beta_2$ . There is a built-in `size` function which returns the size in bits of a binary. That is, `size  $\beta^n = n$ .`

<sup>1</sup>Integers are *not* restricted to those that fit in a single machine word.

$e ::= v$	value
$X$	variable
$\{e_i^{i \in 1..n}\}$	tuple
$e.i$	projection
$\text{fun } X \rightarrow e$	function abstraction
$e_1 e_2$	function application
$\text{let } X = e_1 \text{ in } e_2$	let binding
$\text{letfn } f \ X = e_1 \text{ in } e_2$	recursive function
$v ::= \dots -1, 0, 1 \dots$	integer values
$a, b, c, \dots$	atom values
$\{v_i^{i \in 1..n}\}$	tuple value

Figure 1. Syntax of the core language

$e ::= \langle\langle cs_i^{i \in 1..n} \rangle\rangle$	binary construction
$cs ::= e_v : e_s$	construction segment
$v ::= \#0, \#1, \#00, \#01, \#10, \dots$	binary values

Figure 2. Syntax of binary values and binary construction

## 2.2 Binary construction

We add the syntax for binary construction and for construction segments to our language. This syntax is shown in Figure 2. Informally, each construction segment specifies a sequence of bits and a binary is constructed by concatenating all of the bit sequences defined by the segments in a binary construction expression.

**Construction segments** For a construction segment of the form  $e_v : e_s$  we refer to  $e_v$  as its *value expression* and to  $e_s$  as its *size expression*. In well-typed programs, the value expression evaluates to a binary and the size expression evaluates to a non-negative integer.

**Semantics of binary construction** The reduction rules in Figure 3 state that all of the value and size expressions can be reduced independently and when they have been reduced into base terms the entire binary construction expression can be reduced into a binary.

$\frac{e_v \rightarrow \beta^k \quad e_s \rightarrow n}{e_v : e_s \rightarrow \beta^k : n}$	[EVAL EXPR]
$\frac{\forall i \beta_i^{k_i} = \xi_i^{n_i} \oplus \zeta_i^{k_i - n_i}}{\langle\langle \beta_i^{k_i} : n_i \rangle\rangle \rightarrow \xi_1^{n_1} \oplus \dots \oplus \xi_m^{n_m}}$	[CONSTRUCT]

Figure 3. Reduction rules for binary construction

**Example of binary construction** Let us see how binary construction works. Suppose we are creating two binaries using the following code.

```
let B = <<#0001:4, #0:1>> in <<#101:3, B:(size B)>>
```

Binary construction creates the first binary, with value #00010, and B gets bound to it. Therefore (size B) evaluates to 5 and finally the second binary construction evaluates to #10100010.

Long sequences of zeros and ones are not very reader-friendly. Thus, from now on, for simplicity we will slightly abuse notation and also write integers to the left of the  $:$  symbol. This allows us to rewrite the previous example more simply as:

```
let B = <<1:4, 0:1>> in <<5:3, B:size(B)>>
```

$e ::= \text{case } e \text{ of } p_i \rightarrow e_i^{i \in 1..n} \text{ end}$	case expression
$p ::= v$	constant pattern
$X$	variable pattern
$\{p_i^{i \in 1..n}\}$	tuple pattern
$\langle\langle ps_i^{i \in 1..n} \rangle\rangle$	binary pattern
$ps ::= v : e_s$	literal pattern segment
$X : e_s$	variable pattern segment
$X : *$	tail pattern segment

Figure 4. Syntax additions for binary pattern matching

However, we stress that this mix of integers and binaries is just for the reader's convenience. Integers and binaries are different types.

## 2.3 Binary pattern matching

In addition to constructing binaries it is also important to make it easy for the programmer to decompose them. To allow for this, we introduce a pattern matching case expression in our language. The syntax for this construct is shown in Figure 4. Patterns used in case statements are values, tuples, variables, but also binary patterns whose syntax is also shown in the same figure.

**Matching segments** Assume we are matching a binary pattern  $\langle\langle ps_i^{i \in 1..n} \rangle\rangle$  against a binary  $\beta_m$ . In a pattern segment of the form  $v : e_s$  or  $X : e_s$ , we refer to  $e_s$  as its *size expression*. The first pattern segment  $ps_1$  should be matched to  $s$  bits of  $\beta_m$ , where  $s$  is the result of evaluating the size expression. When the content of the first segment is a variable  $X$  the matching succeeds provided that  $s \leq m$ . This binds  $X$  to a binary consisting of the first  $s$  bits of  $\beta_m$ . When  $ps_1$  has the form  $v : e_s$ ,  $v$  can be either a binary value or, for programming convenience, an integer value. In the case when  $v$  is a binary, the match succeeds when the first  $s$  bits of  $\beta_m$  are equal to  $v$ . When  $v$  is an integer,  $k$ , an explicit conversion of the first  $s$  bits of the binary we are matching out ( $\beta^m$ ) first needs to be performed by the compiler using a conversion built-in called `to_integer`. The matching succeeds if  $k == \text{to\_integer}(\text{first}(s, \beta^m))$ . The process then continues with the remaining pattern segments  $ps_i^{i \in 2..n}$ .

There is also a *tail pattern segment*, having the form  $X : *$ , which can only be the last segment in a binary pattern. This segment binds what remains of  $\beta_m$  to the variable to the left of the  $:$  symbol (in this case  $X$ ).

**Two examples** To show how binary pattern matching works we will give two examples before presenting their semantics formally.

**Example 2.1** The simplest possible matching statement is when we have only one pattern consisting of variables as in the code:

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

The expression  $\langle\langle 8:4, 63:6 \rangle\rangle$  evaluates to #1000111111 and matching this value to the pattern shown in the code results in the bindings  $A = \#1000$  and  $B = \#111111$ .

In the next example, we will display binaries as a sequence of comma-separated 8-bit integers inside  $\langle\langle \rangle\rangle$ 's. This will make it easier to display larger binaries. If the size is not divisible by eight, we will indicate the size of the binary's last component explicitly. For example, for a 20-bit binary consisting of zeros we will write:  $\langle\langle 0, 0, 0:4 \rangle\rangle$ .

**Example 2.2** Assuming some definition for the function `f_bin`, consider the following case statement:

$\frac{}{\text{match}(X, v) = [X \mapsto v]}$	[VARIABLE]
$\frac{v_1 = v_2}{\text{match}(v_1, v_2) = []}$	[VALUE]
$\frac{\forall i \text{ match}(p_i, v_i) \rightarrow \sigma_i}{\text{match}(\{p_i^{i \in 1..n}\}, \{v_i^{i \in 1..n}\}) \rightarrow \sigma_1 \circ \dots \circ \sigma_n}$	[TUPLE]
$\frac{e \rightarrow m}{\text{match}(z : e, ps_i^{i \in 2..n}, \beta^s) \rightarrow \text{match}(z : m, ps_i^{i \in 2..n}, \beta^s)}$	[SIZE EVAL]
$\frac{\beta^s = \beta_1^m \oplus \beta_2^{s-m}}{\text{match}(\langle\langle X : m, ps_i^{i \in 2..n} \rangle\rangle, \beta^s) \rightarrow [X \mapsto \beta_1] \circ \text{match}(\langle\langle ps_i^{i \in 2..n} \rangle\rangle, \beta_2^{s-m})}$	[VARIABLE SEGMENT]
$\frac{\beta^s = \beta_1^m \oplus \beta_2^{s-m} \quad \beta_1^m = \xi^m}{\text{match}(\langle\langle \xi^m : m, ps_i^{i \in 2..n} \rangle\rangle, \beta^s) \rightarrow \text{match}(\langle\langle ps_i^{i \in 2..n} \rangle\rangle, \beta_2^{s-m})}$	[BINARY VALUE SEGMENT]
$\frac{\beta^s = \beta_1^m \oplus \beta_2^{s-m} \quad \text{to\_integer}(\beta_1^m) = k}{\text{match}(\langle\langle k : m, ps_i^{i \in 2..n} \rangle\rangle, \beta^s) \rightarrow \text{match}(\langle\langle ps_i^{i \in 2..n} \rangle\rangle, \beta_2^{s-m})}$	[INTEGER VALUE SEGMENT]
$\frac{}{\text{match}(\langle\langle X : * \rangle\rangle, \beta^s) \rightarrow [X \mapsto \beta^s]}$	[BINARY TAIL SEGMENT]

**Figure 5.** Reduction rules for bit-level binary pattern matching

```

case Binary of
  <<0:8, X:*>> -> f_bin X
  <<X:12, _:12>> -> f_bin X
  <<Sz:8, _:to_integer(Sz), X:*>> -> f_bin X
end.

```

Here Binary will match the pattern in the first branch of the case statement if its first 8 bits converted to an integer have the value 0. In this branch of the case statement, X will be bound to a binary consisting of the rest of the bits of Binary. Otherwise, if Binary is exactly 24 bits long, X will be bound to a binary consisting of the first 12 bits of Binary. If this is not the case, then Binary will match the third pattern if the size of binary is at least  $8 + \text{to\_integer}(\text{Sz})$  bits where  $\text{to\_integer}(\text{Sz})$  is the value of the 8 first bits of Binary converted to an integer. Notice that this is a non-linear binary pattern since the size of one segment depends on a previous segment. This ability to have repeated occurrences of variables in binary pattern matching expressions is not strictly necessary, but we will include it nevertheless since it comes in handy in many applications. Finally, if none of the patterns match, the whole case expression will fail. Three examples of matchings and a failure to match using this code are shown in Table 1.

Binary	Matching of X
$\langle\langle 0, 14, 15, 16, 17 \rangle\rangle$	$\langle\langle 14, 15, 16, 17 \rangle\rangle$
$\langle\langle 1, 2, 20 \rangle\rangle$	$\langle\langle 1, 0 : 4 \rangle\rangle$
$\langle\langle 24, 1, 2, 3, 10, 20 \rangle\rangle$	$\langle\langle 10, 20 \rangle\rangle$
$\langle\langle 128, 255 \rangle\rangle$	<i>failure</i>

**Table 1.** Matchings of X using the code in Example 2.2

**Semantics of binary pattern matching** As formal semantics of binary pattern matching we define the reduction rules for a function `match` in Figure 5. This function takes a pattern and an expression as arguments and either returns a new environment or fails.

The semantics for the case statement is that if  $i$  is the lowest value for which  $\text{match}(p_i, e)$  does not fail then the case expression reduces to  $\text{match}(p_i, e) \circ e_i$ . That is, the expression  $e_i$  is evaluated in the environment returned by  $\text{match}(p_i, e)$ .

Note that the segment reduction rules fix the matching order from left to right for segments. This is necessary since the size expression of a segment can depend on a matching in a previous segment. An instance of such a use was shown in Example 2.2.

### 3. Bit Streams in Erlang

We have implemented the constructs described in Section 2 in the dynamically typed, strict, functional language Erlang. For this reason, the examples in the rest of the paper will be written using Erlang syntax. Whenever there are syntax differences between the core language and Erlang, these will be outlined.

Note that currently Erlang has some constructs for binary construction and pattern matching [14], but those constructs are significantly less general than the ones we present in this paper. In particular, binaries in Erlang only allow for construction and pattern matching against *byte streams* instead of *bit streams*, they are encumbered by some rather ad hoc restrictions, and the semantics of binary pattern matching is defined only by the implementation, not in any formal way.

#### 3.1 Adding conversion specifiers

When pattern matching against binaries, it is convenient to be able to write integers and automatically have them converted to binaries where appropriate. In a typed language — or in a language with a sufficiently powerful type inferencer — this can easily be done by the compiler. In the context of a dynamically typed language, this conversion has to be explicitly requested by the programmer. Thus, we allow for *conversion specifiers* to be included in construction and matching segments. Thus, segments can either have the

Type	Description
binary	No conversion
integer	Converts an integer into a binary by taking the most significant byte first.
integer-little	Converts an integer into a binary by taking the least significant byte first.

**Table 2.** Conversion specifiers for construction segments in Erlang

Type	Description
binary	No conversion
integer	Converts a binary into an integer by putting the first byte in the binary as the most significant byte in the integer and zero extending the value
integer-little	Converts a binary into an integer by putting the first byte in the binary as the least significant byte in the integer and zero extending the value
integer-signed	Converts a binary into an integer by putting the first byte in the binary as the most significant byte in the integer and sign extending the value
integer-signed-little	Converts a binary into an integer by putting the first byte in the binary as the least significant byte in the integer and zero extending the value

**Table 3.** Conversion specifiers for matching segments in Erlang

form  $e_v:e_s$  (in which case a default converter is used) or the form  $e_v:e_s/t$ , where  $t$  is a conversion specifier.

In construction segments, conversion specifiers request a conversion from a prime type of the language to a binary. For Erlang, the allowed conversion specifiers for construction segments are shown and described in Table 2. In matching segments, the conversion specifiers request a conversion from a binary into another language term. Table 3 describes the allowed conversion specifiers for matching segments. To exemplify the meaning of the different conversion specifiers we show the results when using different specifiers in a matching segment in the case statement shown above the table in Figure 6.

case <<0,128>> of <<X:16/t>> -> X	
$t$	X
binary	<<0,128>>
integer	128
integer-little	32768
integer-signed	128
integer-signed-little	-32768

**Figure 6.** Bindings for X for different values of  $t$ .

### 3.2 Default values for segment fields

Since the introduction of conversion specifiers tends to make segments somewhat verbose, it is handy to have some default values for size expressions and conversion specifiers.

The default value for the conversion specifier of a construction segment depends on the type of the value expression. That is, if the type of the value expression is `integer` then the conversion specifier is `integer`. In addition, when the conversion specifier is `integer`, the default value for the size expression is 8. The same default value of 8 is also used for all other integer-based conversion specifiers. On the other hand, when the type of the value expression  $e_v$  is `binary` then the segment type is `binary` and the default value for the size expression is `size( $e_v$ )`.

The default segment type for matching segments is `integer`, and the default size expression is 8 if any of the integer-based conversion specifiers are used. For matching segments with a `binary` conversion specifier, the default size expression is `'*`, which means that they are treated as tail segments.

These rules are probably best understood by seeing the default expansions of various construction and matching segments in some examples. Refer to Table 4.

Construction	Matching	Default expansion
12		12:8/integer
12:S		12:S/integer
<<1,2>>		<<1,2>>:16/binary
X/binary		X:size(X)/binary
	X	X:8/integer
	X:S	X:S/integer
	X/binary	X:*/binary

**Table 4.** Various segments and their default expansions

Conversion specifiers and default expansions allows us to write the program of Example 2.2 in Erlang as follows:

```
case Binary of
  <<0, X/binary>> -> f_bin(X);
  <<X:12/binary, _:12>> -> f_bin(X);
  <<Sz, _:Sz, X/binary>> -> f_bin(X)
end.
```

Note that another minor difference between the core language and Erlang is that we wrap function arguments in parentheses. This is because Erlang does not support currying.

### 3.3 Pattern matching in function clauses

The `case` construct in Erlang is not the only pattern matching facility. Pattern matching is also allowed in clause heads. Since we want binary pattern matching to work exactly like any other pattern matching, we allow for pattern matching on binaries in function clause heads as well. However, this is just syntactic sugar. Pattern matching in function heads can easily be transformed into a case statement as shown in Figure 7.

### 3.4 Bitwise operations on binaries

Finally, since binaries represent sequences of bits, it is natural to define bitwise operations on them such as bitwise-or, bitwise-xor, bitwise-and and bitwise-not. We call these operators `bsor`, `bsxor`, `bsand` and `bsnot`. We show a specification of `bsand` and `bsnot` in Figure 8. Note that `bsand` (and all other bitwise operators on binaries) will fail if its arguments do not have the same size.

## 4. Two Illustrative Examples

To illustrate the power of binary pattern matching at the bit level, we show how some programs which are hard to write in ordinary

<pre> bsand(&lt;&lt;1:1, B1/binary&gt;&gt;, &lt;&lt;1:1, B2/binary&gt;&gt;) -&gt;   &lt;&lt;1:1, bsand(B1, B2)&gt;&gt;; bsand(&lt;&lt;_:1, B1/binary&gt;&gt;, &lt;&lt;_:1, B2/binary&gt;&gt;) -&gt;   &lt;&lt;0:1, bsand(B1, B2)&gt;&gt;; bsand(&lt;&lt;&gt;&gt;, &lt;&lt;&gt;&gt;) -&gt;   &lt;&lt;&gt;&gt;. </pre>	<pre> bsnot(&lt;&lt;1:1, B/binary&gt;&gt;) -&gt;   &lt;&lt;0:1, bsnot(B)&gt;&gt;; bsnot(&lt;&lt;0:1, B/binary&gt;&gt;) -&gt;   &lt;&lt;1:1, bsnot(B)&gt;&gt;; bsnot(&lt;&lt;&gt;&gt;) -&gt;   &lt;&lt;&gt;&gt;. </pre>
--	--

**Figure 8.** The specification of two bitwise operators on binaries — more efficient implementations are possible

```

f(X,Y) ->
  case {X,Y} of
    {p1,1,p1,2} when g1 -> b1;
    {p2,1,p2,2} when g2 -> b2;
    .
    .
    {pn,1,pn,2} when gn -> bn
  end.

```

(a) Function using pattern matching case statement

```

f(p1,1,p1,2) when g1 -> b1;
f(p2,1,p2,2) when g2 -> b2;
.
.
f(pn,1,pn,2) when gn -> bn.

```

(b) Function using pattern matching in function clauses

**Figure 7.** Two equivalent functions

languages can be expressed easily using the bit-level constructs we have introduced.

**Iterating and filtering a bit stream** Consider a variation of the `drop_third` program introduced in [19] that requires inspecting bits besides counting them. The task is to drop all 3-bit chunks that begin with a zero from a bit stream of size evenly divisible by three. Using binary pattern matching this can be written as in Figure 9.

```

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

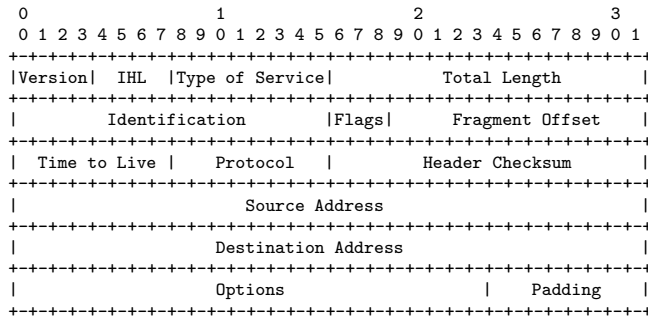
```

**Figure 9.** `drop_0XX` using binaries without size restrictions

The solution is natural as well as 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 binary. The second clause handles the case where the first bit is a zero: we throw that 3-bit chunk away. Finally the last clause handles the case where there are no more chunks: we return the empty binary.

Contrast this with a program written in a language that does not support manipulation of bit streams very well such as C or Java. The programmer would 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 the triple, and calculate how much padding is needed in the output stream. This example shows how binary pattern matching allows the programmer to perform tasks at the specification level. This becomes even more clear when considering how to parse an IP packet using bit-level pattern matching.

**Parsing IP packets** In RFC 791 [16] the IP protocol header is exemplified with the diagram shown in Figure 10. Note the close resemblance between this representation and the pattern shown in Figure 11 which is used to parse an IP version 4 packet header.



**Figure 10.** Internet Protocol Datagram Header, from RFC 791

```

parse_IPv4_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:(32*(IHL-5))/binary,
  Data/binary>>) when Version == 4 ->
  ...

```

**Figure 11.** Function clause parsing an IPv4 packet

Note also that the program 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.

## 5. Binary Comprehensions

Binary comprehensions are expressions that encapsulate recursion patterns on the binary datatype. They are analogous to the widely-used list comprehensions [21], which in turn are expressions that are syntactic sugar for the combination of `map` and `filter` on lists. The main difference between a list and a binary in this case is that what constitutes an element in a list is something *a priori* and unambiguously defined. In contrast, because binaries are terms without (much of a) structure, for binary comprehensions the user must specify what is considered an element of a binary. In particular, the size of elements on which to recurse has to be specified explicitly.

### 5.1 Introductory examples

As a first example of the usefulness of binary comprehensions, let us see how the `bsnot` operator could be implemented using this construct. One possible implementation is the following:

```

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

```

where `bnot/1` is the built-in bitwise Boolean not operator of Erlang for integers. As can be seen, here we consider each bit as

an element in the binary. If we knew the actual element size of the binary, for example that we have a binary whose size is divisible by eight (i.e., a binary which is a sequence of bytes), we could have defined `bsnot` in the following way:

```
bsnot(Bin) ->
  << bnot(X):8 || <<X:8>> <= Bin >>.
```

In short, in a binary comprehension it is both possible and mandatory to specify what is considered an element of the input binary and how the segments of the resulting binary are to be constructed.

The `bsnot` example shows how a binary comprehension can be used to perform a `map` operation on binaries. The following example introduces filtering as well. Consider the `drop_0XX` task of Section 4. It is quite clear that each 3-bit chunk is an element in the binary. If the binary were converted to a list where each element consisted of a 3-bit binary, we would write the following list comprehension to drop the 3-bit binaries starting with a zero:

```
[ <<1:1,X:2>> || <<1:1,X:2>> <- List ]
```

Note that here the binary pattern to the right of the `||` works as a filter as well as a generator; only elements in the list which match the pattern are kept in the output list of 3-bit binaries.

In the previous example the elements were already defined when the list was constructed. For a binary comprehension the elements must be defined in the comprehension. Using binary comprehensions, `drop_0XX` would simply be written as:

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

Notice that this function works in exactly the same way as the function of Figure 9. Here we are forced to wrap the “output” segment in a binary construction because the syntax for binary comprehensions allows for only a single binary segment as output. Using filtering expressions, we can construct only one output segment and write the following version of the `drop_0XX` program which is equivalent to the one above:<sup>2</sup>

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

Sometimes more complicated, perhaps user-defined, filtering is needed. In the following example, we want to remove all non-digit characters from a string and store each of the digits in four bits:

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

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

Where `'$'` is an operator which given a character returns the ASCII value of that character.

## 5.2 Definition of binary comprehensions

The extensions to the syntax of our core language needed to allow for binary comprehensions are shown in Figure 12. The construction segment before the double bars generates a new element in the resulting binary given an environment which is constructed by the qualifiers on the right of the double bars. Two types of qualifiers exist: *binary generators* and *filter expressions*. The binary generator generates a new environment by matching a binary pattern to a binary and each filter expression evaluates to either `true` or `false`. If it evaluates to `false` the current environment is not used to generate a new element, if it evaluates to `true` the environment is used to create a new element.

$e ::= \langle\langle cs \mid q_i^{i \in 1..m} \rangle\rangle$	binary comprehension
$q ::= \langle\langle ps_i^{i \in 1..n} \rangle\rangle \leq e$	binary generator
$\mid e_f$	filter expression

Figure 12. Syntax extensions for binary comprehensions

## 5.3 Semantics of binary comprehensions

The semantics of binary comprehensions are given in terms of their translation into the core language of the paper. Naturally, the translation uses binary pattern matching and binary construction.

To define binary comprehensions in terms of a few simple reduction rules we start by defining  $\_ps_i$  as  $\_ : e$  if  $ps_i = v : e$  and as unchanged otherwise. That gives us a segment  $\_ps_i$  that will always match the incoming binary and consume as many bits as  $ps_i$  from it (i.e., as many as specified by the value of the size expression  $e$ ). Notice that segments containing variables rather than values (i.e., segments of the form  $X : e$ ) are kept unchanged, since variables such as  $X$  might later be used in some size expression. With the help of this definition, the semantics of binary comprehensions are given by the reduction rules of Figure 13.

The binary generator reduction rule in Figure 13(a) works as follows. The first `case` clause attempts matching of the incoming binary with the segments in the binary generator. If this clause does not match, the second clause attempts skipping as many bits as would have been matched. The final clause stipulates that if there are too few bits remaining in the binary to fit the pattern, we should simply throw away these bits. For example consider the `drop_0XX` function applied to a binary of a size that is not evenly divisible by three. For this semantics of binary comprehensions those remaining one or two bits will be left without consideration. An alternative semantics would throw an exception if there are still some remaining bits. The only change in the reduction rules that would be required to have this semantics is to change the third pattern in Figure 13(a) from  $\langle\langle\_ : * \rangle\rangle$  to  $\langle\langle \rangle\rangle$ .

## 5.4 Extended comprehensions

Our binary comprehensions use binaries as generators, but there is no real reason to disallow list generators. This makes it possible to construct binaries from lists in a very flexible way.

For example, consider a list of pairs where the first element contains an integer which represents the number of bits that should be used to encode the integer in the second element. The encoding could easily be performed with the following comprehension:

```
<<X:S || {S,X} <- List>>
```

This is not a made-up example. For example, this kind of situation occurs during Huffman encoding. We call binary comprehensions which allow for both lists and binaries as generators *extended binary comprehensions*. Note that list and binary generators are distinguished by which arrow is used. For list generators `'<-'` is used, whereas `'<='` is used for binary generators. If desired, a statically typed language with a sufficiently powerful type inferencer could get away with using only the `'<-'` symbol by overloading it.

Since we allow for list generators in binary comprehensions it also seems reasonable to allow for binary generators in list comprehensions. This is very useful when trying to convert a binary into a structured term representation. To give a concrete example we show in Program 1 how to collect the filenames and the uncompressed and compressed sizes of all files in a zip-archive [15] using a list comprehension with a binary generator.

We call list comprehensions which also allow binary generators *extended list comprehensions*. We collectively refer to extended list and extended binary comprehensions as *extended comprehensions*. Due to space limitations, a detailed description of the semantics of

<sup>2</sup>In Erlang, `2#100` represents the number four in base two.

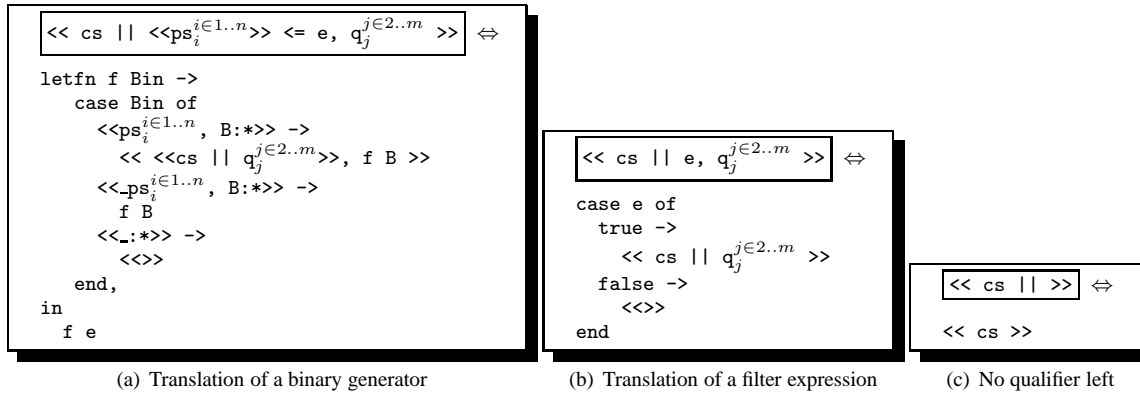


Figure 13. Translation of binary comprehensions into the core language

### Program 1 Extracting file information for files in a zip-archive

```

-module(zip).
-export([collect_fileinfo/1]).

-define(MAGIC, 16#04034b50).
-define(SPEC, integer-little).

collect_fileinfo(ZipBin) ->
  [{FileName,CompSz,UnCompSz} ||
   <<?MAGIC:32/?SPEC, _:80, _Crc32:32/?SPEC,
    CompSz:32/?SPEC, UncompSz:32/?SPEC,
    FileNameSz:16/?SPEC, ExtraSz:16/?SPEC,
    FileName:(8*FileNameSz)/binary, _:(8*ExtraSz),
    _:(8*CompSz)>> <= ZipBin]

```

extended comprehensions is not given here; the interested reader is referred to [7] for their translation to Core Erlang, from which their translation to the language of this paper follows easily.

## 6. Some Applications

We show the usefulness of these operations on binaries by giving examples from application areas where processing of bit streams is ubiquitous. The areas are multimedia processing and network programming. The applications we consider are protocols for digital audio encoding and decoding, picture and file encoding, compression, and processing of telecommunication protocols. Since we have implemented these extensions in the context of the Erlang programming language we will write the examples using Erlang's syntax. However, other than the ability to manipulate binaries at the bit level, there's nothing which is intrinsically Erlang-specific.

### 6.1 $\mu$ -law

Audio files are nowadays transmitted over the network using a variety of formats. One such format, designed to be space efficient, is  $\mu$ -law compressed files [11]. Such files are compressed to half the size of the original audio as each 16-bit sample is translated into an 8-bit representation.

**$\mu$ -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 binary comprehension:

```

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

```

which simply takes each 16-bit sample in 2's complement form and applies the `to_sign_magn` function on it. This function is defined in the following way:

```

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 binary comprehension:

```

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

```

In this case, `S` contains the sign bit and `M` is a binary 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:

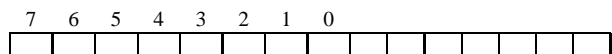
```

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

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

```

What this function does is it searches for the position of the first 1 in the `Magn` binary. Since the range of the magnitude is 132–32766 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:



That is, if the third bit contains the first 1, its position is 5. The following four bits are called the mantissa. In the byte which the `to_byte` function creates 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 which can be done easily using the `bsnot` operator of Section 3.4. The complete code for  $\mu$ -law encoding is shown as Program 2.

**$\mu$ -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 binary comprehension:

```

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

```

---

## Program 2 $\mu$ -law encoding Erlang module

---

```
-module(mu_law).
-export([encode/1]).

encode(Sound) ->
  Biased = << to_sign_magn(Sample) ||
    <<Sample:16/integer-signed>> <= Sound >>,
  Encoded = << to_byte(Sign, Magn) ||
    <<Sign:1,Magn:15/binary>> <= Biased >>,
  bsnot(Encoded).

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

sign(Sample) when Sample >= 0 -> 0;
sign(Sample) when Sample < 0 -> 1.

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

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

---

where `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 can be done with the following 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>>.
```

## 6.2 PNG

The Portable Network Graphics (PNG) file format [18, 12] 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 binary where we have removed the signature from the original file, we can recreate the raw data in order to decompress it using the following binary comprehension.

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

The decimal 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 binary. We can then decompress this data and use the uncompressed data and the chunks containing meta data to generate the picture.

## 6.3 yEnc

To encode a binary file in the yEnc format [9] the binary comprehension in the following program is sufficient:

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

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

where `critical` is a function which returns `true` when a byte needs to be escaped and `false` otherwise. A byte is deemed critical and needs to be escaped if it represents NULL, TAB(ASCII 9), LF(ASCII 10), CR (ASCII 13), or '='.

## 6.4 IS-683 PRL

IS-683 is a telecommunications standard for Over-the-Air Service Provisioning of mobile stations in spread spectrum systems [20]. A PRL is a Preferred Roaming List where one of the records consists of a 5-bit integer followed by a sequence of 11-bit integers. The 5-bit integer describes how many 11-bit integers it is followed by. Each 11-bit integer represents a CDMA channel. Converting this record to a list of CDMA channels is a very simple task using an extended comprehension:

```
decode_prl(<<NumC:5, Channels:(11*NumC), _/binary>>) ->
  [Channel || <<Channel:11>> <= Channels].
```

In short, all these applications can be written rather succinctly using the constructs we introduced. Let us now see whether these constructs can be implemented efficiently.

## 7. Efficient Implementation

### 7.1 Internal representation of binaries

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

Figure 14 shows the representation of a base binary and two sub-binaries. 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 binary (or sub-binary). In the figure, A, B, and C are all variables bound to binaries. A is bound to the base binary <<47, 47, 101, 1:7>>, B is bound to the sub-binary <<22:5>> and C is bound to the sub-binary <<47, 101>>.

### 7.2 Implementation of binary construction

Binary construction is aided by two auxiliary functions:

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

`put_binary()` which given a pointer, an offset in bits, a size in bits, and a binary writes the first size bits of the binary starting at offset bits from the pointer.

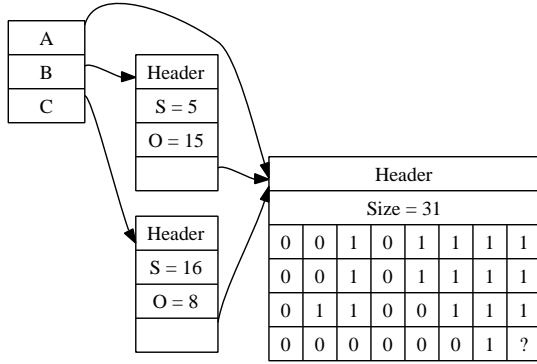


Figure 14. Internal representation of binaries

Using these functions we translate a binary construction expression of the form  $\langle\langle v_{e_1} : s_{e_1} / t_1, \dots, v_{e_n} : s_{e_n} / t_n \rangle\rangle$  as follows. We start by evaluating all the value and size expressions so that we end up with an expression of the form  $\langle\langle v_1 : s_1 / t_1, \dots, v_n : s_n / t_n \rangle\rangle$  where all the  $v_i$ 's are values and all the  $s_i$ 's are non-negative integers. If any  $s_i$  is a negative value, a run-time exception is raised.

Then, we perform the following operations:

1. Calculate the resulting size of the binary as  $\sum_{i=1}^n s_i$ .
2. Allocate a base binary with a large enough data array to hold all of the bits of the binary, 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  $t_i$  is `integer` we call `put_integer(data_ptr, offset, s_i, v_i)`
  - (b) If  $t_i$  is `binary` we call `put_binary(data_ptr, offset, s_i, v_i)`
  - (c) Set `offset` to `offset + s_i`
4. After all segments are processed, return the base binary.

### 7.3 Implementation of binary pattern matching

We only describe the case of matching a binary against a single binary pattern. For a thorough treatment of how to efficiently match a binary against many patterns simultaneously using deterministic binary pattern matching automata refer to our prior work [8].

The matching is aided by two 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_binary()` which creates a sub-binary from an offset, a size and a pointer to a base binary.

To match  $\langle\langle X_1 : e_1 / t_1, \dots, X_n : e_n / t_n \rangle\rangle$  against a binary  $\beta^s$  we perform the binary matching in the manner described below.

1. Create a matching state from  $\beta^s$ .  
This matching state contains the following information:
  - `data_ptr` a pointer to the binary data
  - `offset` the present offset into the binary data
  - `end` the offset of the last bit in the binary
  - `orig_ptr` a pointer to the base binary which contains the data
2. For each segment, perform the following tasks:
  - (a) Evaluate  $e_i$ , the size expression of the first segment to the integer  $s_i$ .
  - (b) Check whether `offset + s_i`  $\leq$  `end`, or else the matching fails.
  - (c) If  $t_i$  is `integer` then  $X_i = \text{get\_integer}(\text{data\_ptr}, \text{offset}, s_i)$

(d) If  $t_i$  is `binary` then  $X_i = \text{get\_binary}(\text{offset}, s_i, \text{orig\_ptr})$

(e) Set `offset` to `offset + 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  $X_n / \text{binary}$ ) is handled specially: we bind  $X_n$  to `get_binary(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  $X_i : e_i / t_i$  where  $X_i$  is a variable. If some  $X_i$  is not a variable but has a value  $v_i$  we simply add an equality test that checks that  $v_i$  is equal to the value returned from either `get_integer` or `get_binary`. If not equal, the matching fails. Otherwise the matching continues with the next segment.

### 7.4 Complexity of binary operations

All data representations have pros and cons. A big advantage of the two structures we use are their simplicity. Another, the fact that they have constant space overhead regardless of a binary's size. But let us also consider some common use cases and see their complexity.

- *Extract one integer of  $m$  bits from a binary of  $n$  bits.* Since a binary is always represented by an area of contiguous bits we can do this in  $O(m)$ .
- *Extract one binary of  $m$  bits from another binary of  $n$  bits.* We can do this easily by creating a sub-binary which is an  $O(1)$  operation.
- *Iterate through a binary of  $n$  bits by extracting an  $m$  bit integer in each iteration.* Since we extract one  $m$  bit integer and create a sub-binary in each iteration, we perform  $n/m$  iterations in total. Thus, this is an  $O(n)$  operation.
- *Construct a binary of  $n$  bits in one binary construction.* Clearly, this is an  $O(n)$  operation.
- *Construct a binary  $B$  by concatenating  $m$  bits to another binary until  $B$  has a size of  $n$  bits.* At each iteration we have a cost of  $i * m$  where  $i$  is the number of iterations. This means that the operation has cost  $O(n^2/m)$ .

In short, all common uses have acceptable complexity but for the last one. However, our simple representation for binaries allows for building efficient abstractions on top of it that in turn make it possible to build binaries piecemeal more efficiently.

### 7.5 Efficient abstractions and alternatives

On top of our representation, abstractions such as creating a *segmented binary* or using a *buffer abstraction* are possible.

**Segmented binaries** A segmented binary consists of a list of binaries and represents the binary that is formed if the binaries in the list are concatenated. This abstraction makes it easy and cheap to concatenate a new binary to an existing segmented binary: all we need to do is to put it first in the list. Then, to efficiently turn a segmented binary into a regular contiguous binary we introduce a built-in called `list_to_binary` which simply transforms a list of binaries into one contiguous binary. This way, constructing a binary of size  $n$  piecemeal from some other binaries can be done in  $O(n)$ .

The problem with segmented binaries is that in the worst case (when each element in the list is a one bit binary) we have a space overhead of several words for every bit in the binary.

**Buffer abstractions** The idea of buffer abstractions is taken from the Lua programming language [10]. A buffer is basically a list of binaries with the following invariant: each binary in the list is strictly bigger than the next binary in the list. Note that, since the representation is a list of binaries, the `list_to_binary` built-in can then be used to turn a buffer into a contiguous binary.

However, since we need to maintain the invariant that binaries in the list are increasing in size, sometimes we need to concatenate binaries directly when adding binaries to the buffer. This makes construction of a buffer more expensive than constructing a segmented binary, but the invariant keeps the space overhead lower for a buffer than for a segmented binary, since the maximal length of the list is  $O(\sqrt{n})$  if the total number of bits is  $n$ .

## 7.6 Implementation of binary comprehensions

The implementation of binary comprehensions requires considering the implications of the chosen underlying representation. If we choose to implement binary comprehension naively using the rewrite rules defined in Section 5.3, the cost of performing the comprehension will be quadratic in the size of the resulting binary.

Naturally we can do better than this. One possible choice is to use segmented binaries, i.e. build a list of binaries and then use `list_to_binary` to convert the list into a binary. This can be expressed very succinctly using extended comprehensions. We can implement a binary comprehension  $\ll\langle cs \mid \mid q^{i \in 1..m} \rangle\rangle$  using an extended list comprehension  $[\ll\langle cs \rangle \mid \mid q^{i \in 1..m}]$  to create a segmented binary which we can then pass to `list_to_binary`.

Another possibility is to collect all of the binaries in a list accumulator and at the same time calculate the sum of the sizes of the binaries in the list. In this way we end up with the size of the resulting binary and a list where the first element in the list contains the binary that should be put last in the result. We can then allocate a large enough base binary and copy the binaries in the list into the data array of that base binary. This would allow us to skip the pass in `list_to_binary` where the size of the resulting binary is calculated.

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

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

```
<<e:se/t \mid \ll\langle e_1:se_1/t_1, \dots, e_n:se_n/t_n \rangle\rangle <= Bin, e_f>>
```

If all of the size expressions ( $se, se_1, \dots, se_n$ ) can be evaluated before the binary comprehension starts being evaluated, then we can calculate how many bits of the input binary 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  $\text{Bin} = \beta^m$  the maximal number of bits in the resulting binary is:

$$(m \times se) / \sum_{i=1}^n se_i$$

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

```
<<42:N \mid \ll\langle S:8,N:(S*S) \rangle\rangle <= Bin>>
```

Luckily such comprehensions are rather rare in practice. Thus, in our implementation we chose to stick to a simple implementation of binary comprehensions, namely that which uses an extended list comprehension flattened by a call to `list_to_binary`. Uncommon cases, such as those where a tight upper bound on the size of the resulting binary cannot be calculated statically, are of course handled correctly, though not optimally.

## 8. A Taste of Performance

Our main goal in this work is to enhance (purely) functional languages with flexible and expressive constructs for bit-level 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 binaries.

Notice however that our binaries are immutable data structures. The language we defined provides no support for destructive assignments to binaries or to their segments. Also, notice that memory management for binaries 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 following programs which spend the bulk of their work in bit stream manipulation.

**drop\_0XX** This is the program from Sections 4 and 5.1. 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 of the task.

**five11** This benchmark is based on the IS-683 PRL example of Section 6.4. A file is read where the first 16 bits in the file represent an integer which describes how many PRL packets the file contains. Each packet starts with a 5-bit integer describing how many channels are in this packet and is followed by that number of 11-bit channel descriptors. The objective of this benchmark is to create a list of channel descriptors for each IS-683 PRL packet. The data we used in the benchmark consists of 496 different packets; 16 of each possible size. These packets are decoded 10,000 times. Manipulating bit chunks whose size is a multiple of five and eleven is not very byte-friendly.

**huffman** The input for this benchmark is a file containing the huffman tree and a message encoded using this tree. The objective of this benchmark is to recreate 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** In this benchmark the objective is to decode a file which 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** In this benchmark the objective is to uuencode 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.

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 versions of the first three benchmarks and found publicly available **uudecode** and **uuencode** C programs on the net which we converted to appropriate benchmarks. 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.

Benchmark	Functional			Imperative
	Erlang	Haskell	O’Caml	C
<b>drop_0XX</b>	2.09	5.85	2.25	1.59
<b>five11</b>	4.97	8.65	7.69	9.78
<b>huffman</b>	2.29	7.38	10.81	0.97
<b>uudecode</b>	3.21	6.04	2.65	0.86
<b>uuencode</b>	2.85	7.77	2.82	1.04

Table 5. Runtimes (in secs)

On the other hand, the imperative languages are free to (and indeed do) use destructive assignments on all benchmarks.<sup>3</sup>

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 Erlang we used the HiPE native code compiler in the pre-release of Erlang/OTP R11. The machine we used is a 2.4 GHz Pentium 4 with 1 GB of memory running Fedora Core 3.

### 8.1 Runtime performance

Table 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 functional 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. Also, 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.

In Table 5 some 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 C 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` for each channel description rather than allocating a big memory area once and partitioning it to each channel using programmer-controlled pointer bumping.

### 8.2 Succinctness and ease of programming

Performance is only part of the story. Another part of the motivation for our work is ease of programming. 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. Table 6 shows that the Erlang solutions are 2–20 times more compact than the solutions in the 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 stream at a suitable granularity with the constructs introduced in this paper.

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. The annotated source code is accessible at <http://bitbenches.infogami.com/>.

<sup>3</sup>The benchmark site also contains Java versions of some of these programs, but some Java results are too embarrassing to be included in this document.

Benchmark	Erlang	Haskell	O’Caml	C
<b>drop_0XX</b>	2	47	45	31
<b>five11</b>	9	38	23	64
<b>huffman</b>	14	30	54	67
<b>uudecode</b>	20	91	65	43
<b>uuencode</b>	25	70	70	54

Table 6. Lines of code

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.

## 9. Related Work

Currently, very few general purpose languages — functional or otherwise — provide constructs for direct manipulation of binary data down at the bit level, let alone efficient ones. Binaries are typically represented as strings (i.e., 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 [13] is such a domain-specific language, designed to ease the implementation of cryptographic applications. It is a functional language based on the notion of sequences of blocks upon which one can perform many different transformations including comprehensions. Cryptol is implemented on top of Haskell and comes with support for pattern matching. However, compared with our work, this support is limited since in Cryptol the constituents that make up the patterns can only have fixed sizes.

SLED [17], a Specification Language for Encoding and Decoding, is a domain-specific language to ease the development of compilers, assemblers, debuggers and profilers which manipulate machine instructions. The language allows the user to describe machine instructions using patterns and constructors. In SLED, *fields* are used to compose patterns, similarly to how segments are used to build patterns in our work, but SLED only allows fields with constant size as opposed to our segments whose size can vary.

Solar-Lezama et al. have recently proposed BitStream, a language for manipulating binaries in the coding and cryptography area [19]. The dataflow programming model used in BitStream is radically different from ours. The language is based on filters, pipelines and split-joins which define transformations on bit streams. The methodology to achieve both correct and efficient programs 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 addition to these domain-specific languages, a series of data description languages have been proposed over the last several years. Many of them allow description of data down at the bit level.

FLAVOR [4] is a data description language for audio-visual applications. Much data in this area is stored in highly compact bit stream formats described by complex standards written in plain text. This makes it hard to know in detail how to encode and decode them. FLAVOR aims to provide a formal description of such formats as well as automatically generate decoding and encoding procedures from a formal description of a bit stream format.

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 down 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 [5] 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 [6] 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. In our work, we propose a syntax for manipulating bit streams in a general purpose functional language. In last year's ICFP, Diatchki, Jones and Leslie [3], proposed a language extension that allows pattern matching on fixed-width bit-data types. Their proposal makes 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.

## 10. Concluding Remarks

The treatment of bit-level data is a neglected area in most general-purpose programming languages; functional languages are no exceptions. In some sense, this is really unfortunate, since there is a plethora of applications out there craving for language constructs that make programs which manipulate bit-streams easier to write and understand, more succinct, less error-prone, while still achieving decent performance.

Our paper has shown that a binary data structure extended with the power to do pattern matching at the bit level and with concepts familiar to functional programmers such as comprehensions allows a functional language to address this need. We see very little reason for bit-level data 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.

Even the limited binaries already included in Erlang have been used in a variety of application areas and programmers have often found innovative uses for them. This, despite the fact that the current binary data type only allows for byte rather than bit streams, that comprehensions on binaries are not yet available, and that the semantics of binary pattern matching are only implementation-defined. Our paper addresses all these issues. At the same time it shows that there is very little which is Erlang-specific in what we do; both at the semantics and at the implementation level.

## Acknowledgments

Many thanks to Xavier Leroy for his suggestion on how to improve the performance of one O'Caml program. Thanks also to Donald Bruce Stewart and members of the Haskell-café mailing list for their help in making our original Haskell programs more efficient.

## References

- [1] G. Back. Datascript — a specification and scripting language for binary data. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference*, 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] A. Eleftheriadis and D. Hong. Flavor: A formal language for audiovisual object representation. In *Proceedings of the 12th Annual ACM International Conference on Multimedia*, pages 816–819. ACM Press, 2004.
- [5] 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 Implementation*, pages 295–304. ACM Press, June 2005.
- [6] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15. ACM Press, Jan. 2006.
- [7] 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. ACM Press, Sept. 2005.
- [8] P. Gustafsson and K. Sagonas. Efficient manipulation of binary data using pattern matching. *Journal of Functional Programming*, 16(1):35–74, Jan. 2006.
- [9] J. Helbing. yEnc: Efficient encoding for Usenet and eMail, June 2002. See also [www.yenc.org](http://www.yenc.org).
- [10] R. Ierusalimsky. *Programming in Lua*. Lua.org, second edition, Mar. 2006.
- [11] International Telecommunication Union. *G.711: Pulse code modulation (PCM) of voice frequencies*. Series G: Transmission Systems and Media, Digital Systems and Networks. Standardization Sector of ITU, Geneva, Switzerland, Nov. 1998.
- [12] Joint ISO/IEC International Standard and W3C Recommendation. Portable network graphics (PNG) specification, W3C/ISO/IEC version, Nov. 2003.
- [13] J. R. Lewis and B. Martin. Cryptol: high assurance, retargetable crypto development and validation. In *MILCOM 2003 2003 IEEE Military Communications Conference*, volume 2, pages 820–825. IEEE, Oct. 2003.
- [14] 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/>.
- [15] PKWARE Inc. Appnote.txt - .zip file format specification, version 6.2.0, Apr. 2004. [www.pkware.com/company/standards/appnote/](http://www.pkware.com/company/standards/appnote/).
- [16] J. Postel. RFC 791: Internet Protocol, Sept. 1981. Obsoletes RFC0760. See also STD0005 Status: STANDARD.
- [17] N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *ACM Trans. Prog. Lang. Syst.*, 19(3):492–524, 1997.
- [18] G. Roelofs. *PNG: The Definite Guide*. O'Reilly and Associates, June 1999. See also [www.libpng.org/pub/png/](http://www.libpng.org/pub/png/).
- [19] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu. 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.
- [20] TIA/EIA/IS-683-A. Over-the-air service provisioning of mobile stations in spread spectrum system, June 1998.
- [21] P. Wadler. List comprehensions. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 7, pages 127–138. Prentice-Hall International, 1987.