Functional Manipulation of Bit Streams *

Per Gustafsson   Konstantinos Sagonas

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

Abstract

Binary (i.e., bit stream) data are omnipresent in computer and network applications but most functional programming languages currently do not provide sufficient support for them. Starting from a language with a built-in binary datatype — albeit a crippled one by various implementation restrictions — we extend it so that it becomes as flexible as lists, which is another ubiquitous datatype in functional programs. We then define various familiar higher-order functions on binaries, show how these allow common tasks from the network and multimedia application domains to be programmed in a compact and concise way, and discuss implementation issues that arise when incorporating these extensions in a functional language.

1. Introduction

Functional programming languages traditionally manipulate objects such as numbers (integers and floats), atoms (sequences of alphanumeric constants), and compound terms such as lists and structures (tuples). Some of them also provide a notation for records that allows abstraction and often (some form of) object oriented-style program development. Erlang supports all these types of objects, but also includes a datatype typically not found in other functional languages: *binaries*.

Binaries were first introduced into Erlang in 1992 to provide an efficient container for object code. Subsequently, it was recognized that binaries can be used in applications that perform extensive I/O, networking TCP/IP-style of communication, in GUI systems, and most importantly in protocol programming which is the bread-and-butter of telecommunication applications. Recognizing the importance of binaries, in 1999, a proposal for a binary datatype was presented in [10] and a revised version of it was subsequently introduced into the Erlang/OTP system in 2000.

To show how binaries are useful in a functional language, consider a relatively simple task: given a bit stream consisting of a sequence of 3-bit chunks, we want to return a bit stream consisting of those 3-bit chunks that start with a one, i.e., drop 3-bit chunks starting with a zero. To do this as conveniently as possible in a functional language we would want the bit stream to be represented as a list of triples. Then we could perform this task using the following program[1] written in the syntax of Erlang:

```
drop_0XX([{1,B2,B3}|Rest]) ->
  [{1,B2,B3}|drop_0XX(Rest)];
drop_0XX([{0,_,_}|Rest]) ->
  drop_0XX(Rest);
drop_0XX([]) ->
  [].
```

or by using a list comprehension simply as:

```
drop_0XX(List) ->
  [{1,B2,B3} || {1,B2,B3} <- List].
```

Although such concise one-line solutions appeal to the heart of most functional programmers, there are at least two problems with this approach. First, symbolic term representation comes with a large space overhead: if we use two words to represent a cons cell and four to represent a 3-tuple, we need six words in total to represent each 3-bit chunk. On a 64-bit machine, this would amount to a use of 384 bits to represent 3 bits of information.[2] Second, the input bit stream is likely to have originated from somewhere else. We either received it from the network or read it from a file, so if we want to manipulate it as a list, we need to transform it to and from this representation.

Ideally, we would like to store the stream of N bits in a format that only requires N bits plus possibly a small constant overhead. Binaries provide such a format. At the same time we would like to be able to program with this format as easily as we do with lists. That is, we would like to write the `drop_0XX` function manipulating binaries as simply as we did for the structured term representation of the bit stream, i.e., with code like the one below:

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

Alas, currently such concise code cannot be written in Erlang. This is both because the analogue of list comprehensions on binaries does not exist and due to various restrictions, imposed by the underlying implementation, which often make code that manipulates binaries quite complicated. In this paper we address these issues.

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

[1] Performance-conscious functional programmers might prefer the version with an accumulator parameter to make it tail-recursive or with the first clause written as:

```
drop_0XX([{1,_,_} = T|Rest]) ->
  [T|drop_0XX(Rest)];
```

which avoids the tuple construction. Throughout this paper, in order to focus on the main issues, we do not worry about such optimizations and do not perform them manually. We instead leave this task to the compiler.

[2] One could possibly think of more efficient symbolic representations of the bit stream, but they are still likely to introduce a substantial space overhead.

Contributions The contributions of this paper are as follows:

- We extend the Erlang binary datatype in various directions to allow manipulation of bit streams to be as convenient and flexible as manipulation of lists without sacrificing efficiency.
- We introduce higher-order functions on binaries such as binary comprehensions and fold-like built-ins and discuss issues related to their efficient implementation.
- We illustrate the usefulness of these extensions by presenting very compact solutions (functional pearls?) to common programming tasks from the network protocol and multimedia processing application domains.

All these issues are described in the context of Erlang, but there is nothing which is particularly Erlang-specific in the core of our proposed language extensions. Also, we are not aware of any work that tries to achieve these goals in another functional language. (The only work that comes mildly close is [9] that describes an API for adding bit streams into Haskell, albeit using the C foreign language interface and without having the ability to perform pattern matching or apply higher-order functions on these bit streams.)

Overview To make the paper self-contained and to set the basis for our proposed extensions, the next section reviews the binary datatype and binary pattern matching as currently implemented in the Erlang/OTP system. Our extensions to binary construction and pattern matching are described in Section 3. Section 4 introduces higher-order operations on binaries and binary comprehensions in particular. We then use these extensions to provide concise solutions to simple tasks from the above-mentioned application domains (Section 5). In Section 6 we discuss issues that arise when implementing the proposed extensions, and the paper ends with some concluding remarks.

2. Binaries as in Erlang/OTP R10B

The binary datatype in Erlang/OTP R10B represents a finite sequence of 8-bit bytes. Two basic operations can be performed on a binary: *creation* of a new binary and *matching* against an existing binary.

2.1 Creation of binaries using the bit syntax

Erlang's bit syntax, described in [5] but see also [10], allows the user to conveniently construct binaries and match these against binary patterns. A bit syntax expression (called a Bin in [5]) is the building block used to both construct binaries and match against binary patterns. A Bin is written with the following syntax:

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

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

Segments Each segment expression has the general syntax:

```
Value:Size/SpecifierList
```

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

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

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

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

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

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

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

Tail of a binary As mentioned, if the `binary` type specifier is used without an explicit size specifier, its size gets expanded to the size `all` by default. In the last segment of a binary this use is similar to the familiar list `cdr` operator since a size of `all` means that the binary is matched against the complete remaining binary (cf. also Example 2.1 below). A segment of type `binary` however, must be a sequence of 8-bit bytes (i.e., have a size evenly divisible by eight).

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

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

Table 2. Some binary segments and their default expansions

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

Table 1. Binary segment specifiers: short description

2.2 Binary matching

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

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

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

Example 2.1 As shown below, binaries are generally displayed as a sequence of comma-separated unsigned 8 bit integers inside `<<>>`'s. The Erlang code:

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

results in the binding `A = 10, B = <<11, 12>>`.[3]

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

`Size` fields of segments are not always statically known. In fact, it is often the case that the value of the size field is decided by the matching of a variable in an earlier segment.

Example 2.2 The Erlang code:

```
<<Sz:8/integer,
  Vsn:Sz/integer,
  Msg/binary>> = <<16,2,154,42>>
```

results in the binding: `Sz = 16, Vsn = 666, Msg = <<42>>`.

Naturally, pattern matching against a binary can occur in a function head or in an Erlang `case` statement just like any other matching operation. The next example shows this.

Example 2.3 Consider the case statement

```
case Binary of
  <<42:8/integer, X/binary>> ->
    handle_bin(X);
  <<Sz:8, V:Sz/integer, X/binary>> when Sz > 16 ->
    handle_int_bin(V, X);
  <<_:8, X:16/integer, Y:8/integer>> ->
    handle_int_int(X, Y)
end.
```

Here `Binary` will match the pattern in the first branch of the `case` statement if its first 8 bits represented as an unsigned integer have the value 42. In this branch of the case statement, `X` will be bound to a binary consisting of the rest of the bits of `Binary`. If this is not the case, then `Binary` will match the second pattern if the first 8 bits of `Binary` interpreted as an unsigned integer have a value greater than 16. Notice that this is a non-linear and guarded binary pattern. Finally, if `Binary` is exactly 32 bits long, `X` will be bound to an integer consisting of the second and third bytes of the `Binary` (taken in big-endian order). If neither of the patterns match, the whole match expression will fail. Three examples of matchings and a failure to match using this code are shown in Table 3.

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

Table 3. Matchings for the code in Example 2.3

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

Example 2.4 If `B` and `L` are unbound variables, the matching:

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

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

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

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

Example 2.5 If `U` and `S` are unbound variables, the code:

```
<<U:8/integer-unsigned>> = <<255>>,
<<S:8/integer-signed>> = <<255>>
```

results in the binding `U = 255, S = -1`.

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

3. Binaries as we want them

The binary syntax greatly simplifies the implementation of network protocols in Erlang. However, sometimes the restrictions on the construction of binaries, currently imposed by the underlying implementation, make the use of binaries cumbersome. Let us again consider the task of dropping all 3-bit chunks that begin with a zero. Ideally, using the binary syntax, one would want to write something like the code in Figure 1.

```
drop_0XX(<<1:1, X:2, Rest/binary>>) ->
  <<1:1, X:2, drop_0XX(Rest)>>;
drop_0XX(<<_:3, Rest/binary>>) ->
  drop_0XX(Rest);
drop_0XX(<<>>) ->
  <<>>.
```

Figure 1. `drop_0XX` using binaries without size restrictions

However, the restriction that binaries (and sub-binaries in them) are of a size which is a multiple of eight currently make such code impossible to write.

Instead, the simplest way that this task can currently be programmed in Erlang/OTP R10B using the binary syntax described in the previous section (i.e., without converting to e.g. a list representation) is shown as Program 1.

Program 1 Drop all 3-bit chunks which start with a zero

```
-module(drop_0XX_R10B).
-export([drop_0XX/1]).

drop_0XX(Bin) ->
  drop_0XX(Bin, 0, 0, <<>>).

drop_0XX(Bin, N1, N2, Acc) ->
  Pad1 = (8 - ((N1+3) rem 8)) rem 8,
  Pad2 = (8 - ((N2+3) rem 8)) rem 8,
  case Bin of
    <<_:N1, 1:1, X:2, _:Pad1, _/binary>> ->
      NewAcc = <<Acc:N2/binary-unit:1, 1:1,X:2, 0:Pad2>>,
      drop_0XX(Bin, N1+3, N2+3, NewAcc);
    <<_:N1, _:3, _:Pad1, _/binary>> ->
      drop_0XX(Bin, N1+3, N2, Acc);
    <<_:N1>> ->
      Acc
  end.
```

As we can see the program becomes quite complicated, since at each construction point the size of binaries has to be evenly divisible by eight. To ensure this, we have to keep track of the number of bits we have consumed and the number of bits that we have kept in order to pad the binaries to an admissible size. Having to do this is not programmer-friendly.[4] More importantly, it subtly undermines the use of the bit syntax for writing high-level specifications of common tasks; programming becomes unnecessarily low-level when there is little reason it should become so. In this particular example, we find Program 1 so revolting, we suggest to our readers to not try to understand it; we just invite them to contrast it with the program in Figure 1.

Another problem with the current restrictions on binaries shows up when performing complex pattern matching. Consider extracting the options from an IP packet. A function which does that, using binaries as in Erlang/OTP R10B, is shown in Figure 2(a). First we

[4] The situation is quite similar to what a C programmer would have to do in order to keep track of which bits to extract from the current byte of the incoming bit stream and how much padding is needed in the output stream.

```
ip_options(IPPacket) ->
  <<4:4, HeaderLength:4, _Rest/binary>> = IPPacket,
  <<Header:HeaderLength/binary-unit:32,
    _Data/binary>> = IPPacket,
  <<4:4, _HeaderLength:4, _RestOfHeader:152,
    Options/binary>> = Header,
  Options.
```

(a) Using binaries as in Erlang/OTP R10B

```
ip_options(IPPacket) ->
  << <<4:4, HeaderLength:4, _RestOfHeader:152,
       Options/binary>>:HeaderLength/binary-unit:32,
     _Data/binary >> = IPPacket,
  Options.
```

(b) Using nested binary patterns

```
ip_options(IPPacket) ->
  <<4:4, HeaderLength:4, _RestOfHeader:152,
    Options:(32*(HeaderLength-5))/binary,
    _Data/binary>> = IPPacket,
  Options.
```

(c) Using a complex size expression

Figure 2. Functions extracting the options from an IPv4 packet

have to find out the length of the IP header. Then the header is extracted from the packet and finally the options are extracted from the header. If nested binary patterns were allowed, we could write this in a much more elegant way as shown in Figure 2(b). Note however that allowing arbitrary complex binary patterns requires a non-trivial level of sophistication from the binary pattern matching compiler; for example, notice the double occurrence of the `HeaderLength` variable. Another, less compiler-challenging, solution to extracting the options from an IP packet is to allow any expression in the size field of a binary segment. Then the `ip_options` function could be written in the manner shown in Figure 2(c).

A final, minor inconvenience with the current implementation of binaries in Erlang/OTP is that the type of a segment must be specified when a binary is created. Consider this piece of code:

```
X = <<1,2,3>>, Bin = <<X,4,5>>.
```

In the current version of the bit syntax this gives rise to a "bad argument" exception. To get the intended effect one is forced to write:

```
X = <<1,2,3>>, Bin = <<X/binary,4,5>>.
```

In binary construction, we lift this restriction and make the type of each segment be the same as the type of the term that the expression evaluates to.[5]

3.1 More flexible binaries: summary of changes

In short, the difference between the binaries as they are currently implemented in Erlang/OTP R10B and the more flexible binaries that we propose in this paper are:

1. A binary (or sub-binary) can have any bit-size, not necessarily one which is divisible by eight.
2. The `Size` field of a segment can contain an arbitrary arithmetic expression (which evaluates to a non-negative integer).
3. No `unit` specifier is needed since `Size` is an arbitrary expression. This allows the user to uniformly specify the size of segments in bits, irrespectively of the segment's type (cf. § 2.1).
4. No type specifier is needed in binary construction.

[5] It is of course an error if an expression evaluates to a term whose type is not one of the allowed types of binary segments.

```
binand(<<1:1, B1/binary>>, <<1:1, B2/binary>>) ->
  <<1:1, binand(B1, B2)>>;
binand(<<_:1, B1/binary>>, <<_:1, B2/binary>>) ->
  <<0:1, binand(B1, B2)>>;
binand(<<>>, <<>>) ->
  <<>>.
```

```
binnot(<<1:1, B/binary>>) ->
  <<0:1, binnot(B)>>;
binnot(<<0:1, B/binary>>) ->
  <<1:1, binnot(B)>>;
binnot(<<>>) ->
  <<>>.
```

Figure 3. A possible implementation of bitwise operators

5. In pattern matching, we allow for nested binary patterns as in: `<<S:16, <<1:8, Bin/binary>>:S, Rest/binary>>`.

How these changes can be implemented in Erlang/OTP is discussed in Section 6. Besides allowing certain tasks to be programmed more conveniently, these changes are also needed for the binary comprehensions introduced in Section 4.1. Having the following operators as language-level built-ins will also come in handy.

3.2 Bitwise operations

Adding built-in bitwise operations on binaries is a natural extension. For example, this allows taking the Boolean `and` of two binaries which have the same size. This is often useful, for example when binaries are used as a bit vector to represent a set. In that case `and`-ing two binaries gives the intersection of two sets.

We propose three different binary operators and one unary operator. The three binary operators are `binand`, `binor`, `binxor` and the unary operator is `binnot`. These four operators can be defined easily using binary matching and construction. We give the definitions of two of them in Figure 3.

4. Higher-order functions on binaries

4.1 Binary comprehensions

Binary comprehensions are expressions that are intended to encapsulate recursion patterns on the binary datatype. They are analogous to the widely-used list comprehensions [8], which in turn are expressions which 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 explicitly specify what is considered an element of a binary.

4.1.1 Introductory examples

As a first example of the usefulness of binary comprehensions, we show how `binnot` could be implemented using this construct. One possible implementation is the following:

```
binnot(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 that the actual element size of the binary is something else, 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 `binnot` in the following way:

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

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

The `binnot` 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 the introduction. 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,B:2>> || <<1:1,B:2>> <- List]
```

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

In this 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,B:2>> || 1:1,B:2 <- Bin>>.
```

Notice that this function works in exactly the same way as the function of Figure 1. 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. Also notice that the ability to create binaries of arbitrary size — of 3 bits in this case — is a prerequisite for flexible binary comprehensions.

Sometimes more complicated, perhaps user-defined, filtering is needed in which case a filter expression is written at the end of the binary comprehension. In the following example, which shows the power both of creating binaries whose size is possibly not a multiple of eight and of using filters in binary comprehensions, we only want to use elements which are in a certain range.

Example 4.1 (UU-decode) If `UUencodedBin` is a binary file that has previously been UU-encoded then we can decode it with this binary comprehension:

```
uudecode(UUencodedBin) ->
  <<(X-32):6 || X <- UUencodedBin, 32=<X, X=<95>>.
```

That is, if the value of a byte is between 32 and 95, we should subtract 32 from that value and put it in the next six bits of the new binary we are creating. (Recall that the default expansion for the segment `X` above is `X:8/integer-unsigned`; cf. also Table 2). If the value is not in that range it is dropped. (This applies to line breaks which are inserted into UU-encoded binaries to make sure that it is possible to display the binary.)

4.1.2 Definition of binary comprehensions

We propose the following syntax for binary comprehensions:

$$\texttt{<<Seg || Seg}_1\texttt{,...,Seg}_k\texttt{ <- Bin, FilterExpr>>}$$

In a binary comprehension we have three distinct parts. One which describes how each new element in the binary shall look, one which describes what we consider as “basic” elements in the input binary, and one which filters these elements. `Seg` is a segment representing the elements of the resulting binary, $\texttt{Seg}_1,\ldots,\texttt{Seg}_k$ are segments

used to describe elements in the old binary Bin, and FilterExpr is a filtering expression used to decide which elements from the old binary should be used to construct the new one. The filtering expression can contain any Boolean expression, or be a sequence of Boolean expressions separated by commas (,). When Boolean expression are separated by commas as in Example 4.1 all of them have to evaluate to the Erlang atom 'true' for the filter to evaluate to true. That is, we can view comma as a shorthand for Boolean and. This is consistent with the syntax and semantics of filtering expressions in list comprehensions in Erlang.

4.1.3 Binary comprehensions with multiple generators

Although our binary comprehensions have filtering capabilities and permit pattern matching in binary generators, the observant reader has no doubt noticed that we have not catered for multiple generators. This ability indeed exists in list comprehensions in Erlang; for example, the following:

```
[{X,Y} || X <- [1,2,3], Y <- [4,5], is_odd(X)]
```

produces the list of pairs: [{1,4},{1,5},{3,4},{3,5}].

There is nothing wrong with multiple generators, but our experience is that they are rarely used in practice. One could possibly conceive of interesting uses for multiple generators in binary comprehensions, so, in the spirit of consistency, expressions like:

```
<<<<X:8,Y:8>> || X <- <<1,2,3>>,
                 Y <- <<4,5>>, is_odd(X)>>
```

producing the binary <<1,4,1,5,3,4,3,5>> should also be allowed. However, in order not to complicate the following section unnecessarily and to focus on the main issues, we will ignore the presence of multiple generators in the rest of this paper.

4.1.4 Translation of binary comprehensions to Erlang code

Binary comprehensions are very handy but, with binaries extended as in Section 3, they are just syntactic sugar, in the same way that list comprehensions are.

To see how they can be expressed in the language, let us rewrite all of the matching segments such that the value fields always contain unbound variables to ease translation. We need to add an extra constraint to FilterExpr for each segment that either contains a bound variable or a constant to preserve the semantics.

That is, let us define $_\mathtt{Seg}_i$ = $\mathtt{Var}_i$:$\mathtt{Size}_i$/$\mathtt{SpecifierList}_i$ if $\mathtt{Seg}_i$ = $\mathtt{Value}_i$:$\mathtt{Size}_i$/$\mathtt{SpecifierList}_i$ and $\mathtt{Value}_i$ is a bound variable or a constant, otherwise $_\mathtt{Seg}_i$ = $\mathtt{Seg}_i$. Let us also define FilterExpr* as (FilterExpr 'and' $\mathtt{Var}_i$ == $\mathtt{Value}_i$) for all i such that $\mathtt{Seg}_i \neq _\mathtt{Seg}_i$. This allows us to rewrite:

```
<<Seg || Seg1,...,Segk <- Bin, FilterExpr>>
```

as

```
<<Seg || _Seg1,...,_Segk <- Bin, FilterExpr*>>
```

which can be implemented in Erlang as shown in Figure 4.

```
Fun = fun(B,F) ->
        case B of
          <<_Seg1,...,_Segk,Rest/binary>> ->
            case FilterExpr* of
              true  -> <<Seg, F(Rest,F)>>;
              false -> F(Rest,F)
            end;
          <<>> ->
            <<>>
        end
      end,
Fun(Bin,Fun).
```

Figure 4. An implementation of binary comprehensions in Erlang

4.2 Fold on binaries

Among the most commonly used higher-order functions on lists are fold-like built-ins (or library functions). These functions are used to successively apply a function to all the elements of a list and get back an often aggregate value. They come in two flavors, foldl and foldr, and quite often are also used in combination with map (i.e., mapfoldl, mapfoldr). In this section, we concentrate on foldl. In Erlang, foldl is defined as follows:

```
foldl(Fun, Acc, []) -> Acc;
foldl(Fun, Acc, [Hd|Tail]) ->
  NewAcc = Fun(Hd, Acc),
  foldl(Fun, NewAcc, Tail).
```

When trying to define the analogue of this function on binaries we are again faced with the following question: what exactly should be considered an element of a binary? Since binaries do not have a predefined uniform structure the way lists do, the most reasonable answer is that it is up to the function parameter to specify what it considers as an element, consume it, and return the remaining binary for later consumption. bfoldl on binaries, the analogue of foldl on lists, is thus defined as:

```
bfoldl(Fun, Acc, <<>>) -> Acc;
bfoldl(Fun, Acc, Bin) ->
  {NewAcc, NewBin} = Fun(Bin, Acc),
  bfoldl(Fun, NewAcc, NewBin).
```

i.e., the function parameter returns a pair consisting of the new accumulator and the remaining binary. This way, it is possible to have a very flexible definition of what an element of the input binary is. In Sections 5.4 and 5.5 we show uses of bfoldl.

5. Applications

To show the usefulness of the new operations on binaries we give some 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, and picture and file encoding and compression.

5.1 μ-law

Audio files are transmitted over the network using a variety of formats. One such format, designed to be space efficient, is μ-law compressed files [3]. Such files are compressed to half the size of the original audio as each 16-bit sample is translated into an 8-bit representation.

5.1.1 μ-law encoding

The encoding method is non-trivial but 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 quite simply with a 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. The to_sign_magn function is defined in the following way:

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

This function 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 as follows:

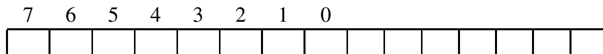
```
<<to_byte(Sign,Magn) || Sign:1,Magn:15/binary <- Biased>>
```

In this case, `Sign` contains the sign bit and `Magn` will be 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 in the following way:

```
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 132–32676 there will be at least one 1 in the first 8 bits. 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 and in the byte that the `to_byte` function creates the first bit contains the sign, the following three contains the position, and the last four bits contain the mantissa.

Finally we take the one complement of this value which can be done easily using the new binary operators of Section 3.2:

```
binnot(Encoded)
```

The complete code for μ-law encoding is shown as Program 2.

Program 2 μ-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>>,
  binnot(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).
```

5.1.2 μ-law decoding

To decode these values we start by taking their 1's complement. After that we 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>>,
```

where `to_short` 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 binary 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>>.
```

5.2 PNG

The portable network graphics (PNG) file format [7, 4] 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 metadata. To recreate the raw data in order to decompress it we can use a simple binary comprehension assuming that the `PNG` variable below is bound to a binary where we have removed the signature from the original file.

```
<<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 metadata to generate the picture.

5.3 yEnc

To encode a binary file in the yEnc format [2] 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 '='.

All the previous examples have shown uses of binary comprehensions; the next two sections show how `bfoldl` can be used.

5.4 Bit sum

One simple common programming task is to find the number of bits which are set in a binary. For example, this allows finding the cardinality of a set when sets are represented using bit vectors. This calculation can be performed by computing the sum of all bits in the binary. That is, if we consider each bit which is set as indicating the presence of an element, taking the sum of all bits gives us the number of elements in the set. If the representation were with a list, this would be a classic case where `foldl` would be used. Thus, it seems appropriate to use `bfoldl` to solve this problem for binaries. Indeed, using `bfoldl` this task can be programmed as simply as:

```
bitsum(Bin) -> bfoldl(fun add_bit/2, 0, Bin).

add_bit(<<X:1,Rest/binary>>, N) -> {N+X,Rest}.
```

5.5 Zip

Zip archives [6] are a commonly used format to store compressed files. Each zip archive starts with a list of local file headers which contains the file data. It subsequently contains some records which describe the directory structure and indicate if encryption has been used or not.

Program 3 returns a pair consisting of the sum of the compressed sizes of all of the files in the zip archive in its first element and the sum of the uncompressed sizes in its second element.

Program 3 Extracting total local file sizes from a zip archive

```
-module(zip).
-export([sizes/1]).

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

sizes(ZipData) ->
  bfoldl(fun add_lfh_size/2, {0, 0}, ZipData).

add_lfh_size(<<?MAGIC:32/?SPEC, _:80, _Crc32:32/?SPEC,
             CompSz:32/?SPEC, UncompSz:32/?SPEC,
             FNameSz:16/?SPEC, ExtraSz:16/?SPEC,
             _:(8*FNameSz), _:(8*ExtraSz), _:(8*CompSz),
             Rest/binary>>, {CS, UCS}) ->
  {{CompSz+CS, UncompSz+UCS}, Rest};
add_lfh_size(_Bin, AccPair) ->
  {AccPair, <<>>}.
```

In this case we only consider local file headers as elements in the binary. As soon as we find something else we are done, as the local file headers are always at the start of the zip archive.

Each local file header structure is quite simple. It starts with a magic number `0x04034b50`. Then there are 10 bytes (80 bits) which are used for flags and to store information about the file. The next four bytes contain a 32-bit cyclic redundancy checksum. After that, we have the size of the compressed file and the size of the uncompressed file. Then there is a two-byte field which describes how many bytes are used to store the file name, followed by another two-byte field which describes the size of the extra field. This is followed by the file name and the extra field and finally we have the compressed file. All the integers in a local file header are stored in little-endian format which explains the use of the `SPEC` macro.

5.6 A very rough comparison

Measuring expressiveness of languages in terms of lines of code is very difficult and often dubious. We nevertheless present in Table 4 the number of lines of code needed to program the core of certain applications manipulating binary data in various languages. (Blank lines, comments, lines of code for performing I/O and memory management are not included.) The chosen languages are C, Java, Erlang as implemented in Erlang/OTP R10B, and Erlang with the extensions and changes proposed in this paper. We do not attempt to draw any general conclusions from such a comparison, but we simply present it as further evidence on the compactness of programs manipulating bit streams using the binary datatype extensions proposed in this paper. We are not aware of other languages where `drop_0XX` or `UU-decode` are essentially one-liners. Appendix A contains sources for the Erlang R10B programs and further information about the origin of the C and Java programs.

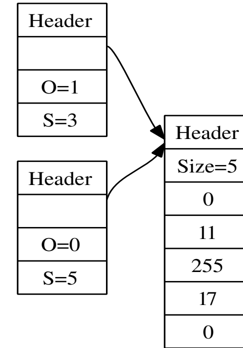


Figure 5. Internal representation of binaries in Erlang/OTP R10B

Program in	C	Java	Erlang (R10B)	Erlang (this)
`drop_0XX`	51	33	14	2
μ-law encode	30	18	25	13
`UU-decode`	19	14	10	2

Table 4. Lines of code needed to process bit stream data

6. Implementation issues

We first describe how binaries and the bit syntax are currently implemented in Erlang/OTP R10B (§ 6.1). We then describe the changes that need to be made in order to implement more flexible binaries (§ 6.2) and efficient binary comprehensions (§ 6.3).

For reference, Table 5 contains a list of some of the instructions of the BEAM (the virtual machine of Erlang/OTP R10B) involving binaries. It should be stated that the description of § 6.1 focuses only on issues relevant to this paper (for example, it does not discuss garbage collection support). For a more complete description of such issues and of how Erlang's bit syntax is translated into BEAM instructions and then into native code see [1].

6.1 Implementation of binaries in Erlang/OTP R10B

Representation of binaries In BEAM, binaries are internally represented using two different constructs:

1. a *binary data block* which contains a header field, a size field and binary data, and
2. a *binary header* which contains a header, a pointer to a binary data block, an offset O, and a size S.

A binary header represents a binary consisting of the bytes starting at byte O in the corresponding binary data block and ending at byte $O + S$ in the data block. (Recall that the the size on bits of binaries in Erlang/OTP R10B is a multiple of eight.) In Figure 5, we show the representation of the binary `<<0,11,255,17,0>>` and its nested sub-binary `<<11,255,17>>`. As shown, several binary headers can point to the same binary data block.

bs_start_match	Calculates auxiliary information needed before the matching starts.
bs_skip_bits	Used in binary pattern matching when the segment to match against involves an anonymous variable.
bs_get_$Type$	Used in binary pattern matching to read a number of bits from the binary and construct a term of type $Type$ from them. Comes in flavors: bs_get_integer, bs_get_float, and bs_get_binary.
bs_put_$Type$	Used in binary construction to turn a term of type $Type$ into a bit stream. After conversion, a number of bits from this stream are written to the binary which is being constructed. Comes in flavors: bs_put_integer, bs_put_float, and bs_put_binary.
bs_init2	Allocates memory for the *binary data block* and returns a pointer to the data block and an initial offset which is zero.
bs_final	Creates the *binary header* from the base pointer and final offset which denotes the total size of the resulting binary.

Table 5. Some BEAM instructions for binaries: short description

Implementation of binary matching A simple binary matching expression of the form:

$$\texttt{<<Seg}_1\texttt{,...,Seg}_n\texttt{>> = Bin.}$$

where each $\texttt{Seg}_i$ is of the general form:

$$\mathsf{Value}_i\mathsf{:Size}_i\mathsf{/Type}_i\mathsf{-Specifiers}_i\texttt{-unit:}\mathsf{Unit}_i$$

is translated in the manner indicated as Translation 1.

Translation 1 Implementation of binary matching in BEAM

$\langle \mathsf{Offset}_0, \mathsf{BinSize}, \mathsf{Base}\rangle = \mathsf{bs_start_match}(\texttt{Bin})$,
$\mathsf{EffSize}_1 = \mathsf{Size}_1 * \mathsf{Unit}_1$
$\langle \mathsf{X}_1, \mathsf{Offset}_1\rangle = \mathsf{bs_get_Type}_1\langle \mathsf{Specifiers}_1\rangle(\mathsf{EffSize}_1, \mathsf{Offset}_0, \mathsf{BinSize}, \mathsf{Base})$,
$\mathsf{Value}_1 = \mathsf{X}_1$
$\vdots$
$\mathsf{EffSize}_n = \mathsf{Size}_n * \mathsf{Unit}_n$
$\langle \mathsf{X}_n, \mathsf{Offset}_n\rangle = \mathsf{bs_get_Type}_n\langle \mathsf{Specifiers}_n\rangle(\mathsf{EffSize}_n, \mathsf{Offset}_{n-1}, \mathsf{BinSize}, \mathsf{Base})$,
$\mathsf{Value}_n = \mathsf{X}_n$
if ($\mathsf{Offset}_n == \mathsf{BinSize}$) Success else Failure

Each $\mathsf{bs_get_Type}_i\langle\mathsf{Specifiers}_i\rangle(\mathsf{EffSize}_i, \mathsf{Offset}_{i-1}, \mathsf{BinSize}, \mathsf{Base})$ instruction reads $\mathsf{EffSize}_i$ bits starting at Offset_{i-1} bits from the Base pointer, if $(\mathsf{Offset}_{i-1}+\mathsf{EffSize}_i) \leq \mathsf{BinSize}$, otherwise the matching fails. The bits that are read are transformed to the Erlang term X_i of type Type_i. How this transformation happens is governed by the $\mathsf{Specifiers}_i$. Finally, Offset_i gets defined as $\mathsf{Offset}_{i-1}+\mathsf{EffSize}_i$.

The binary matching fails either if any of the $\mathsf{bs_get_Type}_i$ instructions fails, or if any of the matchings ($\mathsf{Value}_i = \mathsf{X}_i$) fails, or if we have not matched the complete binary (i.e., Offset_n is not equal to BinSize).

Implementation of binary construction In binary construction we have the following situation:

$$\texttt{NewBin = <<Seg}_1\texttt{,...,Seg}_n\texttt{>>}$$

where again each $\texttt{Seg}_i$ has the general form:

$$\mathsf{Expr}_i\mathsf{:Size}_i\mathsf{/Type}_i\mathsf{-Specifiers}_i\texttt{-unit:}\mathsf{Unit}_i$$

This statement gets translated as indicated in Translation 2.

First, all of the expressions in the value fields are evaluated. Then the size of the resulting binary, BinSize, is calculated from the effective sizes. Then, the binary construction is initialized, a data area that is large enough is allocated, and the pointer to this area is returned along with an Offset_0 which initially is 0. Each bs_put_Type⟨Specifiers⟩(EffSize, Value, Offset, Base) instruction transforms the Value to a bit pattern determined by the Type and the Specifiers. The first EffSize bits of this bit pattern are written to the data area starting at Offset bits from the Base pointer. After the instruction is executed, Offset_i gets defined as $\mathsf{Offset}_{i-1}+\mathsf{EffSize}_i$. Finally, the call to the bs_final instruction creates the new binary from the last Offset and the Base pointer.

Translation 2 Implementation of binary construction in BEAM

$\mathsf{Value}_1 = \mathsf{Expr}_1, \quad \mathsf{EffSize}_1 = \mathsf{Size}_1 * \mathsf{Unit}_1$,
$\vdots \qquad\qquad\qquad \vdots$
$\mathsf{Value}_n = \mathsf{Expr}_n, \quad \mathsf{EffSize}_n = \mathsf{Size}_n * \mathsf{Unit}_n$,
$\mathsf{BinSize} = \mathsf{lists:sum}([\mathsf{EffSize}_1,\ldots,\mathsf{EffSize}_n])$,
$\langle\mathsf{Offset}_0, \mathsf{Base}\rangle = \mathsf{bs_init2}(\mathsf{BinSize})$,
$\mathsf{Offset}_1 = \mathsf{bs_put_Type}_1\langle\mathsf{Specifiers}_1\rangle(\mathsf{EffSize}_1, \mathsf{Value}_1, \mathsf{Offset}_0, \mathsf{Base})$,
$\vdots$
$\mathsf{Offset}_n = \mathsf{bs_put_Type}_n\langle\mathsf{Specifiers}_n\rangle(\mathsf{EffSize}_n, \mathsf{Value}_n, \mathsf{Offset}_{n-1}, \mathsf{Base})$,
$\texttt{NewBin} = \mathsf{bs_final}(\mathsf{Offset}_n, \mathsf{Base})$

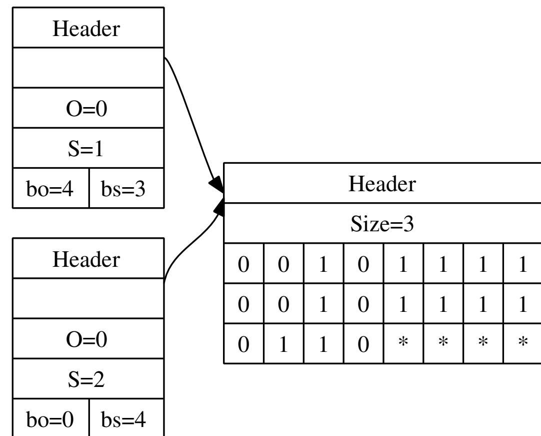


Figure 6. Two binaries with bit sizes which are not multiple of 8

6.2 Implementation of more flexible binaries

Allowing binaries consisting of any number of bits requires few changes to the current representation of binaries. The binary data blocks can remain unchanged, but the binary headers must contain two extra fields *bo* and *bs* which represent a *bit offset* and a *bit size*, respectively. Each of these values ranges from 0-7 and together with the original offset and size fields they represent the offset and size in bits for a binary. That is the exact size in bits of a new flexible binary is $S * 8 + bs$ and the total offset in bits is $O * 8 + bo$.

Figure 6 shows an example of two binaries whose data resides in the same binary data block. One of the binaries consists of all bits in the data block and the other (the one on top) consists of bits 4 to 14 in the data block, i.e., 11110010111.

Allowing arbitrary arithmetic expressions in the size field of a binary segment is also unproblematic and requires rather small changes to the Erlang/OTP compiler. The only modification is to calculate the effective sizes of segments from arithmetic expressions rather than as simply as: $\mathsf{EffSize} = \mathsf{Size} * \mathsf{Unit}$.

On the other hand, removing the need to specify the type during the construction of binary segments slightly complicates binary construction. Partly because it complicates the calculation of a segment's effective size and partly because a new instruction,

$$\mathsf{bs_put_any}\langle\mathsf{Specifiers}\rangle(\mathsf{Effsize},\ \mathsf{Value},\ \mathsf{Offset},\ \mathsf{Base})$$

must be implemented. This instruction determines the runtime type of Value and performs the actions of the appropriate bs_put_$Type$ instruction.

6.3 Implementation of binary comprehensions

The direct translation from binary comprehension to binary pattern matching as shown in Figure 4 is very inefficient. Each time a new segment is added to the binary which is being constructed, the entire binary constructed so far must be copied. This results in quadratic complexity in the number of segments used to create the resulting binary.

One way to avoid the quadratic complexity is to create a binary for each segment, collect these binaries in a list, and at the end of the binary comprehension use this list of binaries to create the resulting binary. The binary comprehension:

<<Seg || _Seg$_1$,...,_Seg$_k$ <- Bin, FilterExpr*>>

would then be source-transformed to the Erlang code of Figure 7. The list_to_binary function is a built-in Erlang function which converts a list into a binary. If its input is list of binaries [b$_1$,...,b$_n$], then its result is the binary <<b$_1$,...,b$_n$>>.

```
Fun = fun(B,F) ->
        case B of
          <<_Seg1,...,_Segk,Rest/binary>> ->
            case FilterExpr* of
              true  -> [<<Seg>>|F(Rest,F)];
              false -> F(Rest,F)
            end;
          <<>> ->
            []
        end
      end,
list_to_binary(Fun(Bin,Fun)).
```

Figure 7. Linear implementation of binary comprehensions

This approach results in a reasonable implementation of binary comprehensions, but it requires that the entire binary is created in small pieces which are then copied into the final binary.

Another approach, which avoids this copying, requires that we first calculate an upper limit on the final size of the resulting binary. We can then allocate a memory area for the *binary data block* and write the segments to the data block as they are created. When all segments have been written, we can create the *binary header* for the resulting binary and adjust the size of the binary data block appropriately. The implementation of this scheme is not a matter of a simple source transformation of the binary comprehension to Erlang code anymore, but needs to be done as part of the translation to BEAM bytecode. We show this for the case where all size expressions of segments evaluate to constants. The translation of a binary comprehension of the form:

<<Seg || _Seg$_1$,...,_Seg$_k$ <- Bin, FilterExpr*>>

is shown as Translation 3. Each segment Seg of the resulting binary is of the general form Expr:SizeExpr/Specifiers, while each matching segment _Seg$_i$ is of the form Var$_i$:SizeExpr$_i$/Type$_i$-Specifiers$_i$. Because all size expressions evaluate to constants, we can simply calculate an upper limit on the size of the resulting binary by the formula:

$$\mathsf{SizeLimit} = \frac{\mathtt{size(Bin)}}{\sum_{i=1}^{k} \mathsf{SizeExpr}_i} \times \mathsf{SizeExpr}$$

and leave it up to the bs_final instruction to adjust the size of the memory area for the binary data block.

In those rare cases where size expressions do not evaluate to constants, as e.g. in:

<<42:N || S:8,N:(S*S) <- Bin>>

and it is not possible to calculate a tight upper limit on the size of the resulting binary when the binary comprehension is encountered, one can either employ a more involved translation scheme or simply use the translation of Figure 7.

7. Concluding remarks

The treatment of binary files, and bit-level data structures in general, is a neglected area in functional languages. With only few notable exceptions, Erlang being one of them, most functional languages do not provide direct support in terms of a concrete datatype and a syntax for performing pattern matching on bit streams. We do not really see any good reason for Erlang to continue being an exception in providing such support; perhaps this paper paves the way towards this direction.

Since their introduction in Erlang, binaries have been heavily used in a variety of applications and programmers have often found innovative uses for them. The extensions to the binary datatype presented in this paper make binaries flexible and the higher-order functions we propose make programming involving binaries more concise and more "functional" in style. We have every reason to believe that, in programs manipulating bit stream data, binary comprehensions will eventually become as common as list comprehensions are in programs which manipulate lists.

References

[1] P. Gustafsson and K. Sagonas. Native code compilation of Erlang's bit syntax. In *Proceedings of ACM SIGPLAN Erlang Workshop*, pages 6–15. ACM Press, Nov. 2002.

[2] J. Helbing. yEnc: Efficient encoding for Usenet and eMail, June 2002. See also www.yenc.org.

[3] International Telecommunication Union. *G.711: Pulse code modulation (pcm) of voice frequencies.* Series G: Transmission Sytems and Media, Digital Sytems and Networks. Standardization Sector of ITU, Geneva, Switzerland, Nov. 1998.

[4] Joint ISO/IEC International Standard and W3C Recommendation. Portable network graphics (PNG) specification, W3C/ISO/IEC version, Nov. 2003.

[5] 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/.

[6] PKWARE Inc. Appnote.txt - .zip file format specification, version 6.2.0, Apr. 2004. Available at: www.pkware.com/company/standards/appnote/.

[7] G. Roelofs. *PNG: The Definite Guide*. O'Reilly and Associates, June 1999. See also www.libpng.org/pub/png/.

[8] 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.

Translation 3 Generic translation of binary comprehensions into BEAM bytecode when size expressions evaluate to constants

```
      EffSize_1 = SizeExpr_1      % evaluates the SizeExpr
        ⋮
      EffSize_n = SizeExpr_n
      ElementSize = lists:sum([EffSize_1,...,EffSize_n])
      EffSize = SizeExpr
      SizeLimit = (size(Bin) / ElementSize) * EffSize
      ⟨MOffset, BinSize, MBase⟩ = bs_start_match(Bin)
      ⟨COffset, CBase⟩ = bs_init2(SizeLimit)
1:
      if (MOffset == BinSize) goto 2
      ⟨Var_1, MOffset⟩ = bs_get_Type_1⟨Specifiers_1⟩(EffSize_1, MOffset, BinSize, MBase)
        ⋮
      ⟨Var_n, MOffset⟩ = bs_get_Type_n⟨Specifiers_n⟩(EffSize_n, MOffset, BinSize, MBase)
      if (FilterExpr(Var_1,...,Var_n) == false) goto 1
      Value = Expr(Var_1,...,Var_n)
      COffset = bs_put_any⟨Specifiers⟩(Value, EffSize, COffset, CBase)
      goto 1
2:
      NewBin = bs_final(COffset, CBase)
```

Program	Taken from
μ-law encode in C	http://www.speech.cs.cmu.edu/comp.speech/Section2/Q2.7.html
μ-law encode in Java	http://www.developer.com/java/other/article.php/3286861
`uudecode.c`	http://www.koders.com/c/fidFDC554B9F9F6340C894141C53BE08C5299D93B5F.aspx
UUDecoder.java	http://www.cs.duke.edu/csed/java/src1.3/sun/misc/UUDecoder.java

Table 6. Origin of the C and Java programs of Table 4

[9] M. Wallace and C. Runciman. The bits between the lambdas: Binary data in a lazy functional language. In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pages 107–117, New York, N.Y., Oct. 1998. ACM Press.

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

A. Additional programs

For completeness, we include in this appendix the complete source code for UU-decode (Program 4) and μ-law encode (Program 5) written using binaries as in Erlang/OTP R10B. Table 6 shows the origin of some of the C and Java programs of Table 4. (The C and Java programs for `drop_0XX` are the only ones written by us.)

Program 4 UUdecoding module in Erlang/OTP R10B

```
-module(uu_R10B).
-export([decode/1]).

decode(EncodedBin) ->
  decode(EncodedBin, 0, <<>>).

decode(<<X, Rest/binary>>, N, Acc) when 32=<X, X=<95 ->
  NewN = N+6,
  Pad = (8-(NewN rem 8)) rem 8,
  NewAcc = <<Acc:N/binary-unit:1, (X-32):6, 0:Pad>>,
  decode(Rest, NewN, NewAcc);
decode(<<_, Rest/binary>>, N, Acc) ->
  decode(Rest, N, Acc);
decode(<<>>, _N, Acc) ->
  Acc.
```

Program 5 μ-law encoding module in Erlang/OTP R10B

```
-module(mu_law_R10B).
-export([encode/1]).

encode(Bin) ->
  Biased = bias_and_sign_magn(Bin),
  Encoded = encode_to_byte(Biased),
  binnot(Encoded).

bias_and_sign_magn(<<Sample:16, Rest/binary>>) ->
  <<sign(Sample):1, (min(abs(Sample), 32635)+132):15,
    bias_and_sign_magn(Rest)/binary>>;
bias_and_sign_magn(<<>>) -> <<>>.

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

encode_to_byte(<<Sign:1, Magn:15, Rest/binary>>) ->
  Exp = exp_lut(Magn bsr 7),
  Mant = (Magn bsr (Exp+3)) band 15,
  <<Sign:1, Exp:3, Mant:4, encode_to_byte(Rest)/binary>>;
encode_to_byte(<<>>) -> <<>>.

exp_lut(X) when X <   2 -> 0;
exp_lut(X) when X <   4 -> 1;
exp_lut(X) when X <   8 -> 2;
exp_lut(X) when X <  16 -> 3;
exp_lut(X) when X <  32 -> 4;
exp_lut(X) when X <  64 -> 5;
exp_lut(X) when X < 128 -> 6;
exp_lut(X) when X < 256 -> 7.

binnot(<<X, Rest/binary>>) ->
  <<bnot(X), binnot(Rest)/binary>>;
binnot(<<>>) -> <<>>.
```