Bit-level Binaries and Generalized Comprehensions in Erlang [*]

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. Erlang is an exception since it does support direct manipulation of binary data, albeit currently restricted to byte streams, not bit streams. To ameliorate the situation, we extend Erlang's built-in binary datatype so that it becomes flexible enough to handle bit streams properly. To further simplify programming on bit streams we then show how binary comprehensions can be introduced in the language and how binary and list comprehensions can be extended to allow both binary and list generators.

Categories and Subject Descriptors   D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms   Design, Languages

Keywords   Erlang, bit-streams, comprehensions, binaries

1. Introduction

Most functional programming languages have support for manipulating objects such as numbers (integers and floats), atoms (sequences of alphanumeric constants), and compound terms such as lists and structures (tuples). Some also provide a notation for records that allows abstraction and often (some form of) object oriented-style program development. However, most of these languages lack facilities for directly manipulating raw streams of bits and bytes.

Erlang is a functional language that breaks the mold in that, in addition to the datatypes described above, it also has a datatype which can represent these streams directly: *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 [5] and a revised version of it was subsequently introduced into the Erlang/OTP system in 2000.

The syntax that was introduced made it easy to handle streams of bytes in Erlang. For example consider the following simple task: given a stream consisting of 3-byte chunks we want to return a stream consisting of those 3-byte chunks whose first byte is zero. This can be written in the following manner:

```
keep_0XX(<<0:8,X:16,Rest/binary>>) ->
  <<0:8,X:16,keep_0XX(Rest)/binary>>;
keep_0XX(<<_:24,Rest/binary>>) ->
  keep_0XX(Rest);
keep_0XX(<<>>) ->
  <<>>.
```

Now consider that instead of a byte stream we want to do the same task with a bit stream of 3-bit chunks. We would then like it to be possible to write a program like this:

```
keep_0XX(<<0:1,X:2,Rest/binary>>) ->
  <<0:1,X:2,keep_0XX(Rest)/binary>>;
keep_0XX(<<_:3,Rest/binary>>) ->
  keep_0XX(Rest);
keep_0XX(<<>>) ->
  <<>>.
```

Unfortunately, in Erlang/OTP R10B this program would not compile, let alone work. This is due to the restriction that all binaries need to have a bit-size which is evenly divisible by eight. We want to lift this restriction to extend Erlang's facilities for dealing with bit streams so as to match the support for handling byte streams.

To see how we can further simplify bit stream programming, let us consider how we would perform the task described above using structured terms. To do this as conveniently as possible we would want the bit stream to be represented as a list of 3-tuples. Then we could perform this task using the following program:

```
keep_0XX([{0,B2,B3}|Rest]) ->
  [{0,B2,B3}|keep_0XX(Rest)];
keep_0XX([{1,_,_}|Rest]) ->
  keep_0XX(Rest);
keep_0XX([]) ->
  [].
```

or by using a list comprehension simply as:

```
keep_0XX(List) ->
  [{0,B2,B3} || {0,B2,B3} <- List].
```

With such a short and elegant solution why would we not use structured terms to perform this task? Notice that there are at least two problems with this solution. First, the structured 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. 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 of triples, we need to transform it to and from this representation.

What we want to do instead is to extend the facilities for manipulating streams in Erlang in such a way as to make it possible to write an equally concise solution which operates directly on bit streams. That is we want it to be possible to write the `keep_0XX` function manipulating binaries as concisely as we did for the structured term representation of the bit stream, i.e., with code like the one below:

```
keep_0XX(Bin) ->
    <<<<0:1,B:2>> || 0:1,B:2 <<- Bin>>.
```

This will be achieved partly by allowing binaries whose bit-sizes are not evenly divisible by eight and partly by introducing binary comprehensions for binaries. The latter are analogous to list comprehensions for lists.

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 show how Erlang built-in-functions that deal with binaries can be extended to handle extended binaries.
- We generalize the concept of comprehensions from list comprehensions to list and binary comprehensions which can use both list and binary generators.

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. How this extensions influence common Erlang built-ins is discussed in Section 4. Section 5 introduces binary comprehensions. We then show in Section 6 how these comprehensions can be extended and combined. In Section 7 we show how extended binary comprehensions can be implemented efficiently, and the paper ends with some concluding remarks.

2. Binaries as in Erlang/OTP R10B

The binary datatype in Erlang/OTP R10B represents a stream 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 [2] but see also [5], allows the user to conveniently construct binaries and match these against binary patterns. A bit syntax expression (called a Bin in [2]) is the building block used to both construct binaries and match against binary patterns. A Bin is written with the following syntax:

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

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:

$$\texttt{Value:Size/SpecifierList}$$

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. Some of 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. The specifiers for signedness and endianess are not described in this paper, but a description of these specifiers can be found in [1]. If all type specifiers are used, the syntax of each segment expression is:

$$\texttt{Value:Size/Type-Signedness-Endianess-unit:Unit}$$

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

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. The default specifiers for an integer segment are a size of 8 bits, and a unit of 1.
- The `float` type specifier only allows effective sizes of 32 or 64 bits. The default specifiers for a float segment are a size of 64 bits, and a unit of 1.
- The `binary` specifier allows effective sizes that are evenly divisible by 8. 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.

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. 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 binary segment however, must 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-unit:1
X/float	X:64/float-unit:1
X/binary	X:all/binary
X:Size/binary	X:Size/binary-unit:8

Table 2. Some binary segments and their default expansions

2.2 Binary matching

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

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

The $\texttt{Value}_i$ fields of the $\texttt{Segment}_i$ expressions that describe each segment will be matched to the corresponding segment in `Binary`. For example, if the $\texttt{Value}_1$ field in $\texttt{Segment}_1$ contains an unbound

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.
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

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 $Segment_1$.

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>>`.

Here `A` matches the first 8 bits of `Binary`. Because of the default values (cf. Table 2), these eight bits are interpreted as an 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 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 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`. 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

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 keeping only 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.

```
keep_0XX(<<0:1, X:2, Rest/binary>>) ->
  <<0:1, X:2, keep_0XX(Rest)>>;
keep_0XX(<<_:3, Rest/binary>>) ->
  keep_0XX(Rest);
keep_0XX(<<>>) ->
  <<>>.
```

Figure 1. keep_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 Keep all 3-bit chunks which start with a zero

```
-module(keep_0XX_R10B).
-export([keep_0XX/1]).

keep_0XX(Bin) ->
  keep_0XX(Bin, 0, 0, <<>>).

keep_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, 0:1, X:2, _:Pad1, _/binary>> ->
      NewAcc = <<Acc:N2/binary-unit:1, 0:1, X:2, 0:Pad2>>,
      keep_0XX(Bin, N1+3, N2+3, NewAcc);
    <<_:N1, _:3, _:Pad1, _/binary>> ->
      keep_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 divis-

```
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:(32*(HeaderLength-5))/binary,
    _Data/binary>> = IPPacket,
  Options.
```
(b) Using a complex size expression

Figure 2. Functions extracting the options from an IPv4 packet

ible 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.[1] 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.

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 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. A simple 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(b).

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:

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

Even though Erlang is a dynamically typed language, in the current version of the bit syntax, the code above 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.[2]

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).

---

[1] 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.

[2] 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.

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.

4. Adapting BIFs to handle flexible binaries

Several Erlang built-in functions already operate on binaries. These built-ins must be extended to handle the new flexible binaries since they can consist of any number of bits, not only those whose bit size is a multiple of eight. In this section the most frequently used built-ins which handle binaries are discussed.

4.1 binary_to_list(Bin)

This built-in constructs a list of values between 0 and 255 where the first element holds the value of the first byte of the binary Bin, the second element holds the value of its second byte, *etc*. The built-in succeeds as long as Bin is a binary.

It can be defined in Erlang in the following way:

```
binary_to_list(<<X:8,Rest/binary>>) ->
  [X|binary_to_list(Rest)];
binary_to_list(<<>>) ->
  [].
```

With this definition, binary_to_list(Bin) will fail if Bin has a size in bits which is not evenly divisible by eight. This is not appropriate; binary_to_list(Bin) should always succeed as long as Bin is a binary. Furthermore, the invariant:

$$Bin == list\_to\_binary(binary\_to\_list(Bin))$$

is important and should be preserved. This leads us to define binary_to_list(Bin) as follows:

```
binary_to_list(<<X:8,Rest/binary>>) ->
  [X|binary_to_list(Rest)];
binary_to_list(<<>>) ->
  [];
binary_to_list(Bin) ->
  [Bin].
```

That is if Bin has a bit size which is not evenly divisible by eight, the function returns a list whose elements are the bytes of Bin and its last element is a binary consisting of the remaining bits. Using this definition the following call to the built-in:

```
binary_to_list(<<0:20>>)
```

returns the list [0,0,<<0:4>>].

4.2 size(Bin)

This built-in function returns the size of the binary Bin in bytes. We need to define what size(Bin) should return in case the size of the binary in bits is not evenly divisible by eight. Our choice is to have size(Bin) return the minimal number of bytes needed to represent the binary. That is for a binary which consists of 20 bits like the <<0:20>> above, size(<<0:20>>) returns 3. This is because the 20-bit binary needs three bytes to be represented.

It would however be necessary to introduce a new built-in called bitsize(Bin) which returns the size of a binary in bits (in our example 20).

5. Examples of 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 [4], 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.

As a first example of the usefulness of binary comprehensions, we show how `binnot` a function which inverts a binary 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 `keep_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 keep the 3-bit binaries starting with a zero:

```
[<<0:1,B:2>> || <<0:1,B:2>> <- List]
```

Note that here the binary pattern to the right of `||` 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, `keep_0XX` would simply be written as:

```
keep_0XX(Bin) ->
    <<<<0:1,B:2>> || 0:1,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 comprehensions allows for only a single 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.

It is also important to understand what would happen if `Bin` is a binary whose bit size is not evenly divisible by three. In this case we would get a matching error and the binary comprehension would raise an exception. Why this is the case is evident since this function works in exactly the same as the function shown in Figure 1 and a 1-bit or a 2-bit binary does not match any of the clauses in that function.

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 and ignore the rest.

Example 5.1 (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.)

6. Extended comprehensions

The binary comprehensions that we introduced in the previous section 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 more flexible way than using `list_to_binary`.

Consider for example a situation where we have a list consisting of pairs where the first element contains an integer which represent the number of bits that should be used to encode the integer contained in the second element. Then we could write the following comprehension:

$$<<X:S \mid\mid \{S,X\} <- List>>$$

This kind of situation could occur during Huffman coding for example. We call binary comprehensions which allow both lists and binaries as generators *extended binary comprehensions*.

It also seems reasonable to allow binary generators in list comprehensions. This is very useful when trying to convert a binary format into a structured term representation. To give a concrete example we show in Program 2 how to collect the filenames and the uncompressed and compressed sizes of all files in a zip-archive [3] using a list comprehension with a binary generator.

Program 2 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) ->
  [binary_to_list(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]
```

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.

6.1 Extended comprehensions with multiple generators

Although our extended 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)]
```

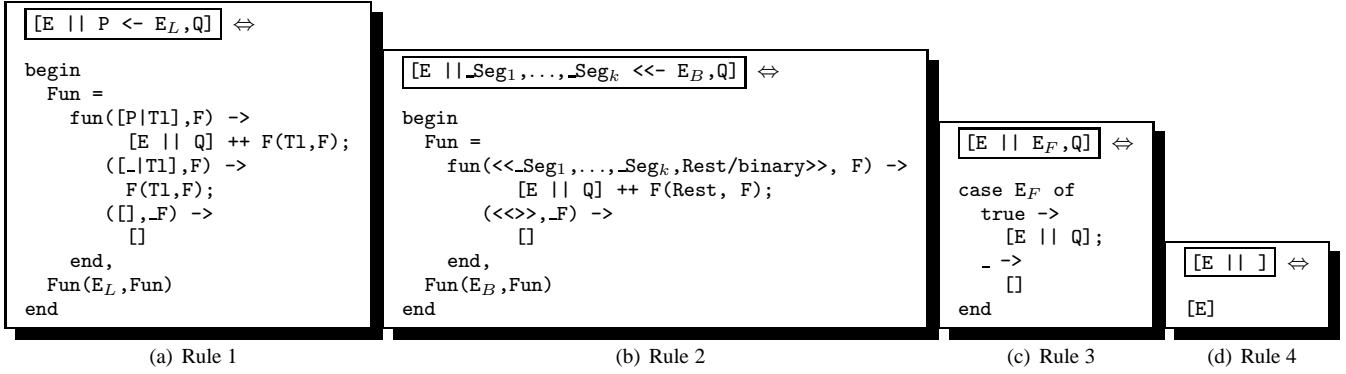produces the list of pairs: `[{1,4},{1,5},{3,4},{3,5}]`.

```
[E || P <- E_L,Q]  ⇔

begin
  Fun =
    fun([P|Tl],F) ->
          [E || Q] ++ F(Tl,F);
       ([_|Tl],F) ->
          F(Tl,F);
       ([],_F) ->
          []
    end,
  Fun(E_L,Fun)
end
```
(a) Rule 1

```
[E ||_Seg_1,...,_Seg_k <<- E_B,Q]  ⇔

begin
  Fun =
    fun(<<_Seg_1,...,_Seg_k,Rest/binary>>, F) ->
          [E || Q] ++ F(Rest, F);
       (<<>>,_F) ->
          []
    end,
  Fun(E_B,Fun)
end
```
(b) Rule 2

```
[E || E_F,Q]  ⇔

case E_F of
  true ->
    [E || Q];
  _ ->
    []
end
```
(c) Rule 3

```
[E || ]  ⇔

[E]
```
(d) Rule 4

Figure 3. Reduction rules for extended list comprehensions

```
<<S || P <- E_L,Q>>  ⇔

begin
  Fun =
    fun([P|Tl],F) ->
          <<<<S || Q>>, F(Tl,F)>>;
       ([_|Tl],F) ->
          F(Tl,F);
       ([],_F) ->
          <<>>
    end,
  Fun(E_L,Fun)
end
```
(a) Rule 5

```
<<S ||_S_1,...,_S_k <<- E_B,Q>>  ⇔

begin
  Fun =
    fun(<<_S_1,...,_S_k,Tl/binary>>, F) ->
          <<<<S || Q>>, F(Tl,F)>>;
       (<<>>,_F) ->
          <<>>
    end,
  Fun(E_B,Fun)
end
```
(b) Rule 6

```
<<S || E_F,Q>>  ⇔

case E_F of
  true ->
    <<S || Q>>;
  _ ->
    <<>>
end
```
(c) Rule 7
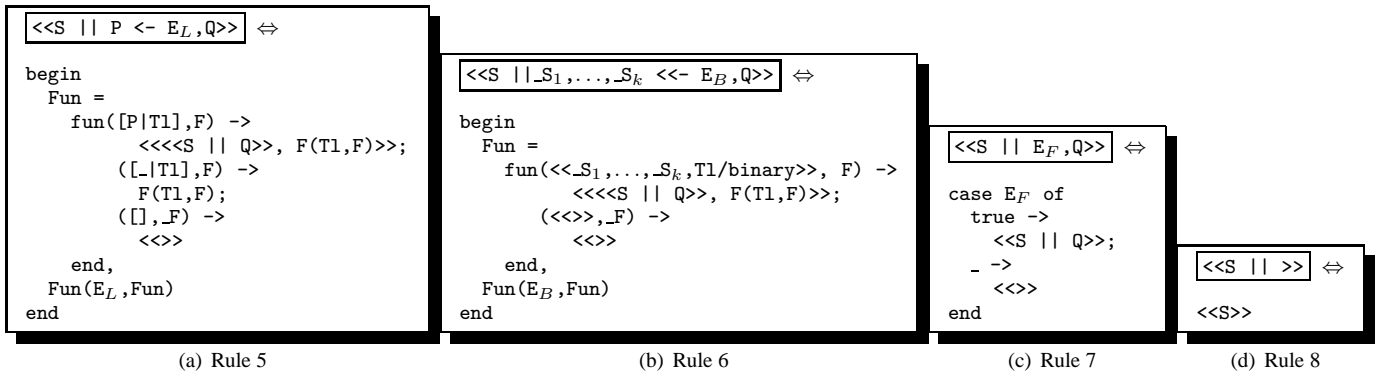
```
<<S || >>  ⇔

<<S>>
```
(d) Rule 8

Figure 4. Reduction rules for extended binary comprehensions

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 extended 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. Our translation in the next section caters for this.

6.2 Semantics of extended comprehensions

To formalize the semantics of extended comprehensions in Erlang we will show how these extended comprehensions can be translated into Erlang code. The syntax for extended list comprehensions is: `[E||Q]` where E is an expression and Q is a comma-separated list of zero or more qualifiers. A qualifier is either a list generator, a binary generator or a filter expression. The syntax for a list generator is P `<-` E_L where P is a pattern and E_L is an expression. The syntax for a binary generator is $S_1,...,S_k$ `<<-` E_B where $S_1,...,S_k$ are segments and E_B is an expression. A filter expression E_F is simply an ordinary Erlang expression. It either evaluates to `true` or to something else which means `false`. For a program to be type correct E_L must always evaluate to a list and E_B must evaluate to a binary.

In order to simplify the handling of binary generators, which were written using the general form:

$$S_1,...,S_k \text{ <<- } E_B$$

let us define a segment $\_S_i$ = `Var_i:Size_i/SpecifierList_i` if S_i = `Value_i:Size_i/SpecifierList_i` and `Value_i` is a bound

variable or a constant, otherwise $\_S_i$ = S_i. Let us also define `FilterExpr` as `Var_i == Value_i` for all i such that $S_i \neq \_S_i$. This allows us to rewrite a binary generator as:

$$\_S_1,...,\_S_k \text{ <<- } E_B, \text{ FilterExpr}$$

When binary generators are rewritten in this manner the reduction rules shown in Figure 3 can be used to translate extended list comprehensions into Erlang code.

The syntax for extended binary comprehensions is: `<<S||Q>>` where S is a segment and Q is a comma-separated list of qualifiers. The qualifiers are the same as the qualifiers for list comprehensions. Rewriting binary generators in the same way as described above we can translate binary comprehensions into Erlang code using the reduction rules shown in Figures 4.

Note that the rules in Figure 3(d) and 4(d) where there are no qualifiers in the comprehensions are not likely to be very common in code, but we allow them to make the description of the semantics more uniform.

To show how these rules can be used to translate a comprehension into Erlang code consider the following comprehension:

```
<<(X+Y):16 || X:16 <<- Bin, Y <- List, X>Y>>
```

Using rule 6 we can transform that into the code shown in Figure 5. The comprehension:

```
<<(X+Y):16 || Y <- List, X>Y>>
```

can then be reduced using rule 5. Doing this we get the code shown in Figure 6.

Using reduction rule 7 and then rule 8 on the comprehension: `<<(X+Y):16 || X>Y>>` we get the code shown in Figure 7.

```
begin
  Fun1 =
    fun(<<X:16,Tl/binary>>, F1) ->
        <<<<(X+Y):16 || Y <- List, X>Y>>,
          F1(Tl, F1)>>;
      (<<>>, _F1) ->
        <<>>
    end,
  Fun1(Bin, Fun1)
end
```

Figure 5. Code produced by applying reduction rule 6 on `<<(X+Y):16 || X:16 <<- Bin, Y <- List, X>Y>>`

```
begin
  Fun2 =
    fun([Y|Tl],F2) ->
        <<<<(X+Y):16 || X>Y>>,
          F2(Tl, F2)>>;
      ([_|Tl], F2) ->
        F2(Tl, F2);
      ([], _F2) ->
        <<>>
    end,
  Fun2(List, Fun2)
end
```

Figure 6. Code produced by applying reduction rule 5 on `<<(X+Y):16 || Y <- List, X>Y>>`

```
case X>Y of
  true ->
    <<(X+Y):16>>;
  _ ->
    <<>>
end
```

Figure 7. Code produced by applying reduction rule 7 and 8 on `<<(X+Y):16 || X>Y>>`

Putting it all together we get the final result which does not use any comprehensions shown in Figure 8.

Fresh names need to be used for the closures when translating comprehensions with this method. This is necessary since Erlang lacks support for recursive closures.

It would be possible to have a different semantics for binary comprehensions where the last couple of bits would just be skipped. The only changes to the reduction rules that need to be made for them to have this semantics would be to change the second clause in reduction rules 2 and 6 from `(<<>>, F)` into `(_, F)`.

7. Implementation

Implementing extended comprehensions using the reduction rules introduced in Sect. 6.2 would be very inefficient since constructing the resulting list or binary would be quadratic in their respective sizes.

We will present a simple translation scheme for extended binary comprehension into Erlang code which avoids the quadratic complexity cost for constructing the resulting binary.

First we translate an extended binary comprehension: `<<S||Q>>` into `list_to_binary(<*S||Q*>)`. `<*S||Q*>` is a comprehension used only in this compilation scheme which has the property that it

```
begin
  Fun1 =
    fun(<<X:16,Tl/binary>>, F1) ->
        <<begin
            Fun2 =
              fun([Y|Tl],F2) ->
                  <<case X>Y of
                      true ->
                        <<(X+Y):16>>;
                      _ ->
                        <<>>
                    end,
                    F2(Tl, F2)>>;
                ([_|Tl],F2) ->
                  F2(Tl,F2);
                ([], _F2) ->
                  <<>>
              end,
            Fun2(List,Fun2)
          end,
          F1(Tl, F1)>>;
      (<<>>, _F1) ->
        <<>>
    end,
  Fun1(Bin,Fun1)
end.
```

Figure 8. Final code when all comprehensions have been reduced in `<<(X+Y):16 || X:16 <<- Bin, Y <- List, X>Y>>`

```
begin
  Fun1 =
    fun(<<X:16,Tl/binary>>, F1) ->
        [<*X || X < 256*>| F1(Tl,F1)];
      (<<>>, _F1) ->
        []
    end,
  Fun1(Bin,Fun1)
end
```

Figure 9. Code produced by applying reduction rule 10 on the comprehension `<*X || X:16 <<- Bin, X < 256*>`

always produces a possibly nested list of binaries. It can be reduced using the reduction rules shown in Figure 12.

A small example will show what kind of code we will end up with using this approach. Consider the comprehension:

$$<<X || X:16 <<- Bin, X < 256>>$$

This comprehension would first be translated into:

`list_to_binary(<*X || X:16 <<- Bin, X < 256*>)`

This can be reduced using reduction rule 10 into the code shown in Figure 9. The comprehension that is left in that piece of code: `<*X || X < 256*>` can be translated into the expression shown in Figure 10 and the complete resulting code is shown in Figure 11.

8. Concluding remarks

The treatment of binaries, and bit-level data structures in general, is a neglected area in functional languages. The only notable exception that we are aware of is the bit syntax in Erlang. The extensions to the binary datatype presented in this paper make binaries flexible and the extended comprehensions we propose make pro-
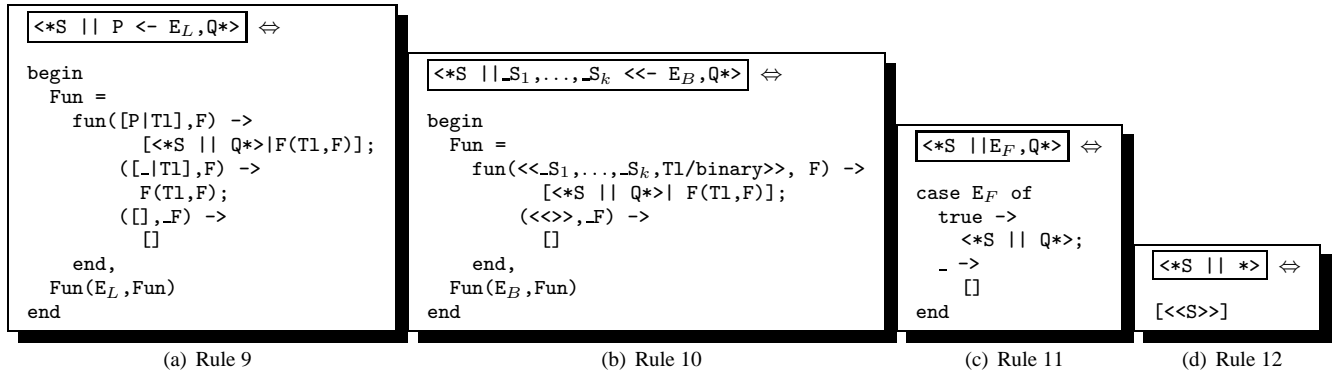
```
<*S || P <- E_L,Q*>  ⇔

begin
  Fun =
    fun([P|Tl],F) ->
          [<*S || Q*>|F(Tl,F)];
        ([_|Tl],F) ->
          F(Tl,F);
        ([],_F) ->
          []
    end,
  Fun(E_L,Fun)
end
```

(a) Rule 9

```
<*S ||_S_1,...,_S_k <<- E_B,Q*>  ⇔

begin
  Fun =
    fun(<<_S_1,...,_S_k,Tl/binary>>, F) ->
          [<*S || Q*>| F(Tl,F)];
        (<<>>,_F) ->
          []
    end,
  Fun(E_B,Fun)
end
```

(b) Rule 10

```
<*S ||E_F,Q*>  ⇔

case E_F of
  true ->
    <*S || Q*>;
  _ ->
    []
end
```

(c) Rule 11

```
<*S || *>  ⇔

[<<S>>]
```

(d) Rule 12

Figure 12. Reduction rules for temporary comprehensions

```
case X < 256 of
  true ->
    [<<X>>];
  _ ->
    []
end
```

Figure 10. Code produced by applying reduction rule 11 and 12 on the comprehension `<<X || X < 256>>`

```
list_to_binary(
  begin
    Fun1 =
      fun(<<X:16,Tl/binary>>, F1) ->
            [case X < 256 of
               true ->
                 [<<X>>];
               _ ->
                 []
             end | F1(Tl,F1)];
          (<<>>, _F1) ->
            []
      end,
    Fun1(Bin,Fun1)
  end)
```

Figure 11. Code produced when reducing all comprehensions in `<<X || X:16 <<- Bin, X < 256>>`.

gramming 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.

We are currently discussing with the OTP implementation team how the changes proposed in this paper can be incorporated into Erlang/OTP. We believe that we will be able to add the changes introduced here as an experimental feature in Erlang/OTP R11.

References

[1] P. Gustafsson and K. Sagonas. Adaptive pattern matching on binary data. In *Programming Languages and Systems: 13th European Symposium on Programming, ESOP 2004, Proceedings*, pages 124–139. Springer, Mar. 2004.

[2] 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/.

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

[4] 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.

[5] 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/.