Bit-level Binaries and Generalized Comprehensions in Erlang

Per Gustafsson and Kostis Sagonas
Dept of Information Technology
Uppsala University
Binaries as we know them

Introduced in 1992 as a container for object code

Used in applications that do I/O, networking or protocol programming

A proposal for a binary datatype and a syntax was made in 1999 and a revised version was adopted in 2000

Since then, binaries have been used extensively, often providing innovative solutions to common telecom programming tasks
Binaries are not so flexible

Some limitations:

- Binaries are byte streams, not bit streams
- Segment sizes cannot be arbitrary arithmetic expressions

Both undermine the use of the binary syntax for writing high level specifications

This work:

We show how to lift these limitations while maintaining backward compatibility
Make binaries as flexible as lists

- In lists:
  - deconstructing a list always yields valid terms
  - can be constructed using list comprehensions

- In binaries:
  - deconstructing a binary sometimes yields terms which cannot be represented as Erlang binaries
  - no binary comprehensions are available

- This work:
  - allows binaries to represent bit streams
  - introduces binary comprehensions
  - introduces extended comprehensions to make conversions between lists and binaries simpler
Flexible bit-level binaries

• The multiple-of-eight size restriction is lifted

• The size field of a segment can contain an arbitrary arithmetic expression

• No type specifier is needed in binary construction
Pros and cons of bit-level binaries

+ Allows natural representation of bit fields
  • `<<BitSize:8, BitField:BitSize/binary, ...`

+ Helps avoid padding calculations
  • `Pad = (8 - ((X + Y) rem 8)) rem 8`,

+ Makes binary matching as easy for bit streams as it was for byte streams
  - Introduces a speed trade-off
Pattern Matching  
- byte streams vs bit streams

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(<<>>) ->  
  <<>>.

This function only keeps the byte triples whose first byte is 0.

But what if we want to keep the bit triples whose first bit is 0?
Pattern Matching
- byte streams vs bit streams

```
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(<<<>) ->
  <<<>.
```

This is how it ought to look!
Pattern Matching
- byte streams vs bit streams

```
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.
```

This is how you have to write it today!
Allowing arithmetic expressions in the size field

Consider this classic example of the bit syntax:

case IP_Packet of
  <<4:4, Hlen:4, SrvcType:8, TotLen:16, ID:16, Flgs:3, FragOff:13, TTL:8, Proto:8, SrcIP:32, DestIP:32, RestDgrm/binary>> ->
    OptsLen = Hlen - 5,
    <<Opts:OptsLen/binary-unit:32, Data/binary>> = RestDgrm,
  ...
end
Allowing arithmetic expressions in the size field

Using flexible binaries it could be written in the following manner:

case IP_Packet of
  <<4:4, Hlen:4, SrvcType:8, TotLen:16,
   ID:16, Flgs:3, FragOff:13, TTL:8,
   Proto:8, SrcIP:32, DestIP:32,
   Opts:((Hlen - 5)*32)/binary,
   Data/binary>> -> ...
end,
No need for a type-specifier in binary construction

Consider the following code:

\[
\begin{align*}
X &= \langle\langle 1, 2, 3 \rangle\rangle, \\
B &= \langle\langle X, 4, 5 \rangle\rangle
\end{align*}
\]

It causes a runtime exception. To avoid this you must explicitly specify the type

\[
\begin{align*}
X &= \langle\langle 1, 2, 3 \rangle\rangle, \\
B &= \langle\langle X/binary, 4, 5 \rangle\rangle
\end{align*}
\]

We want to lift this restriction, the type should default to the type of the variable.
Binary Comprehensions

Analogous to List Comprehensions

List Comprehensions represent a combination of map and filter

Comprehensions require a notion of an element

For binary comprehensions the user must specify what they consider as an element
Binary Comprehensions:
Introductory Example, invert

Using list comprehension:

```python
invert(ListOfBits) ->
[bnot(X) || X <- ListOfBits]
```

Using binary comprehension:

```python
invert(Binary) ->
<<bnot(X):1 || X:1 <- Binary>>
```

If your binary is byte-sized:

```python
invert(Binary) ->
<<bnot(X):8 || X:8 <- Binary>>
```
Using a binary comprehension UU-decode basically becomes a one-liner in Erlang

```
uudecode(UUBin) ->
   <<(X-32):6 || X:8 <- UUBin, 32=<X, X=<95>>
```

Note the filter expressions which make sure that inserted characters such as line-breaks are dropped
Extended comprehensions

Can we use list generators in binary comprehensions?

`convert_to_binary(ListofWords) -> <<X:32 || X <- ListofWords>>.`

YES!
Extended comprehensions

Can we use binary generators in list comprehensions?

```
convert_to_listofwords(Binary) -> [X || X:32 <- Binary].

YES!
```
Generators

Note that we need to be able to separate list generators from binary generators.

List generators:

\[ P \leftarrow E_L \]

Binary generators:

\[ S_1 \ldots S_n \leftarrow E_B \]

- \( P \) – a pattern
- \( E_L \) – an Erlang expression which evaluates to a list
- \( S_i \) – a binary segment
- \( E_B \) – an Erlang expression which evaluates to a binary
Implementation of extended binary comprehensions

• We present a simple translation of extended comprehensions into Erlang in the form of rewrite rules in the paper

• Using these simple rules the cost of building the resulting binary is quadratic in the number of segments

• We present another set of rewrite rules which gives linear complexity, but the rules are slightly less straight-forward
Implementation of extended binary comprehensions

When the size of the resulting binary can be calculated as a function of a generator binary, the translation can be very efficient

\[
\text{Res} = L \ll X:16 \mid X:8 \leq \text{Bin} >. \\
\Rightarrow \\
\text{bitsize(Res)} = (\text{bitsize(Bin)} / 8) \times 16
\]

This allows us to preallocate the memory that is needed for the resulting binary
Example: IS-683 PRL

**Data Structure**

<table>
<thead>
<tr>
<th></th>
<th>5 bits</th>
<th>11 bits</th>
<th></th>
<th>11 bits</th>
<th></th>
<th>11 bits</th>
<th>0-7 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel 1</td>
<td></td>
<td></td>
<td>Channel I</td>
<td></td>
<td>Channel N</td>
<td></td>
<td>Pad</td>
</tr>
</tbody>
</table>

**Task:**
Create a list of Channels
First "Padding" Solution:

decode(<<NumChans:5, _Pad:3, _Rest/binary>> = Bin) ->
decode(Bin, NumChans, NumChans, []).

decode(_, _, 0, Acc) ->
Acc;
decode(Bin, NumChans, N, Acc) ->
  SkipBefore = (N - 1) * 11,
  SkipAfter = (NumChans - N) * 11,
  Pad = 8 - ((NumChans * 11 + 5) rem 8),
  <<_:5, _:SkipBefore, Chan:11,
    _:SkipAfter, _:Pad>> = Bin,
  decode(Bin, NumChans, N - 1, [Chan | Acc]).
Correct "Padding" Solution:

```prolog
decode(<<NumChans:5, _Pad:3, _Rest/binary>> = Bin) ->
    decode(Bin, NumChans, NumChans, []).

decode(_, _, 0, Acc) ->
    Acc;
decode(Bin, NumChans, N, Acc) ->
    SkipBefore = (N - 1) * 11,
    SkipAfter = (NumChans - N) * 11,
    Pad = (8 - ((NumChans * 11 + 5) rem 8)) rem 8,
    <<__:5,__:SkipBefore, Chan:11,
       __:SkipAfter,__:Pad>> = Bin,
    decode(Bin, NumChans, N - 1, [Chan | Acc]).
```
decode(Channels) ->
cast Channels of
<0:3:3,11:2>>
[X1,X2,X3];

Expanded solution:

<<3:5,X1:11,X2:11,X3:11,_,2>> ->
[X1,X2,X3];
Smart, but inefficient solution

```plaintext
decode(<<N_channels:5, Alignment_bits:3, Tail/binary>>) ->
    decode2(N_channels, <<Alignment_bits:3, Tail/binary, 0:5>>).

decode2(0, _) ->
    [];
decode2(N, <<C:11, A:5, T/binary>>) ->
    [C | decode2(N-1, <<A:5, T/binary, 0:3>>)].
```

Avoids complicated padding calculations, at the cost of recreating the binary in each iteration.
Using Flexible binaries

Since flexible binaries can represent bit streams properly and leads to a natural solution

\[
\text{decode}(<\text{N:5, Channels:}(11*\text{N})/\text{binary},_/\text{binary}>>) \rightarrow \\
\text{decode2} (\text{Channels}).
\]

\[
\text{decode2}(<\text{C:11, T/binary}>>) \rightarrow \\
[\text{C}|\text{decode2}(\text{T})]; \\
\text{decode2}(<\text{>>}) \rightarrow \\
[].
\]
Extended comprehensions

Using extended comprehensions and flexible binaries we can solve the problem in two lines:

```
decode(<<N:5, Channels: (11*N)/binary, _/binary>>) ->
[X || X:11 <= Channels].
```
Succinctness of flexible binaries
- as measured in line counts

<table>
<thead>
<tr>
<th>Program in</th>
<th>C</th>
<th>Java</th>
<th>Erlang (R10B)</th>
<th>Erlang (this)</th>
</tr>
</thead>
<tbody>
<tr>
<td>keep 0XX</td>
<td>51</td>
<td>33</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>μ-law encode</td>
<td>30</td>
<td>25</td>
<td>25</td>
<td>13</td>
</tr>
<tr>
<td>UU-decode</td>
<td>19</td>
<td>14</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

- μ-law encode - Compresses sound
- keep 0XX - Keeps bit-triples that start with 0
- UU-decode - Decodes UU-encoded binaries
Conclusion

• Introducing bit-level binaries makes it easy to represent bit streams as binaries
• This makes it possible to write high level specifications of operations on bit streams
• Extended comprehensions allow for powerful manipulation of binaries
• Together these extensions make binaries as easy to use as other datatypes in Erlang such as tuples and lists
• The extensions we propose are backwards compatible
• They will probably be included in the R11 release of Erlang/OTP
Future Work

• A standard library for dealing with binaries

• A better representation of binaries to avoid quadratic complexity when appending binaries

• New compilation techniques which allow for in-place updates of binaries
Adapting BIF:s to bit-level binaries

size(Bin)
  • should return the minimal number of bytes needed to represent the binary.

bitsize(Bin)
  • new bif which returns the size in bits

binary_to_list(Bin)
  > the following should hold:
  \[ \text{Bin} == \text{list_to_binary}(\text{binary_to_list}(\text{Bin})) \]
binary_to_list(Bin)

Desired property:

$$\text{Bin} == \text{list_to_binary}(\text{binary_to_list}(\text{Bin}))$$

$$\text{binary_to_list}(<\ll X:8, \text{Rest}/\text{binary}>>) \rightarrow [X|\text{binary_to_list} \text{(Rest)}];$$
$$\text{binary_to_list}(<<?>>) \rightarrow [];$$
$$\text{binary_to_list}(\text{Bin}) \text{ when is_binary}(\text{Bin}) \rightarrow [\text{Bin}].$$

gives:

$$[0,0,<\ll 0:4>>] == \text{binary_to_list}(<<0:20>>)$$